SuperLU Users' Guide

James W. Demmel* John R. Gilbert† Xiaoye S. Li‡

February 4, 1997

Contents

1 Introduction **3**
- 1.1 About SuperLU . . . 3
- 1.2 Availability . . . 3
- 1.3 How to call a SuperLU routine . . . 3

2 Matrix data structures **5**

3 Permutations **8**
- 3.1 Ordering for sparsity . . . 8
- 3.2 Partial pivoting with threshold . . . 10

4 User-callable routines **10**
- 4.1 Driver routines . . . 11
- 4.2 Computational routines . . . 11

5 Matlab interface **12**

6 Memory management for L and U **13**

7 Installation **14**
- 7.1 File structure . . . 14
- 7.2 Testing . . . 16
- 7.3 Performance-tuning parameters . . . 17
- 7.4 Error handling . . . 17

8 Statistics **18**

*Computer Science Division, University of California, Berkeley, CA 94720 (demmel@cs.berkeley.edu). The research of Demmel and Li was supported in part by NSF grant ASC–9313958, DOE grant DE–FG03–94ER25219, UT Subcontract No. ORA4466 from ARPA Contract No. DAAL03–91–C0047, DOE grant DE–FG03–94ER25206, and NSF Infrastructure grants CDA–8722788 and CDA–9401156.

†Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304 (gilbert@parc.xerox.com). The research of this author was supported in part by the Institute for Mathematics and Its Applications at the University of Minnesota and in part by DARPA Contract No. DABT63-95-C0087.

‡National Energy Research Scientific Computing (NERSC), Lawrence Berkeley National Lab, 1 Cyclotron Rd, Berkeley, CA 94720 (xiaoye@nersc.gov). Part of the work was done while being a graduate student at U.C. Berkeley.

9 Example programs **18**
9.1 Repeated factorizations . . . 20
9.2 Calling from Fortran . . . 20

10 Acknowledgement **20**

A Specifications of routines **23**
A.1 SGSEQU . . . 23
A.2 SGSCON . . . 24
A.3 SGSRFS . . . 25
A.4 SGSSV . . . 27
A.5 SGSSVX . . . 28
A.6 SGSTRF . . . 34
A.7 SGSTRS . . . 37
A.8 SLAQGS . . . 38

1 Introduction

1.1 About SuperLU

The SuperLU package contains a set of subroutines to solve sparse linear systems $AX = B$. Here A is a square, nonsingular, $n \times n$ sparse matrix, and X and B are dense $n \times nrhs$ matrices, where $nrhs$ is the number of right-hand sides and solution vectors. Matrix A need not be symmetric or definite; indeed, SuperLU is particularly appropriate for matrices with very unsymmetric structure.

The package uses LU decomposition with partial pivoting, and forward/back substitutions. The columns of A may be preordered before factorization (either by the user or by SuperLU); this preordering for sparsity is completely separate from the factorization. To improve backward stability, we provide working precision iterative refinement subroutines [2]. Routines are also available to equilibrate the system, estimate the condition number, calculate the relative backward error, and estimate error bounds for the refined solutions. We also include a Matlab MEX-file interface, so that our factor and solve routines can be called as alternatives to those built into Matlab. The LU factorization routines can handle non-square matrices, but the triangular solves are performed only for square matrices.

The factorization algorithm uses a graph reduction technique to reduce graph traversal time in the symbolic analysis. We exploit dense submatrices in the numerical kernel, and organize computational loops in a way that reduces data movement between levels of the memory hierarchy. The resulting algorithm is highly efficient on modern architectures. The performance gains are particularly evident for large problems. There are "tuning parameters" to optimize the peak performance as a function of cache size. For a detailed description of the algorithm, see reference [4].

SuperLU is implemented in ANSI C, and must be compiled with a standard ANSI C compiler. It includes versions for both real and complex matrices, in both single and double precision. The file names for the single-precision real version start with letter "s" (such as `sgstrf.c`); the file names for the double-precision real version start with letter "d" (such as `dgstrf.c`); the file names for the single-precision complex version start with letter "c" (such as `cgstrf.c`); the file names for the double-precision complex version start with letter "z" (such as `zgstrf.c`).

1.2 Availability

The package can be obtained from Netlib through the URL address:

```
http://www.netlib.org/scalapack/prototype/
```

It is also available on the FTP server at UC Berkeley:

```
ftp ftp.cs.berkeley.edu
login: anonymous
ftp> cd /pub/src/lapack/SuperLU
ftp> binary
ftp> get superlu_1.0.tar.gz
```

1.3 How to call a SuperLU routine

As a simple example, let us consider how to solve a 5×5 sparse linear system $AX = B$, by calling a driver routine `dgssv`. Figure 1 shows matrix A, and its L and U factors.

$$\begin{pmatrix} s & & u & u & \\ l & u & & & \\ & l & p & & \\ & & & e & u \\ l & l & & & r \end{pmatrix} \qquad \begin{pmatrix} 19.00 & & 21.00 & 21.00 & \\ 0.63 & 21.00 & -13.26 & -13.26 & \\ & 0.57 & 23.58 & 7.58 & \\ & & & 5.00 & 21.00 \\ 0.63 & 0.57 & -0.24 & -0.77 & 34.20 \end{pmatrix}$$

Original matrix A Factors $F = L + U - I$

$s = 19, u = 21, p = 16, e = 5, r = 18, l = 12$

Figure 1: A 5×5 matrix and its L and U factors.

The program first initializes the three arrays, `a[]`, `asub[]` and `xa[]`, which store the nonzero coefficients of matrix A, their row indices, and the indices indicating the beginning of each column in the coefficient and row index arrays. This storage format is called compressed column format, also known as Harwell-Boeing format [5]. Next, the two utility routines `dCreate_CompCol_Matrix` and `dCreate_Dense_Matrix` are called to set up matrices A and B, respectively, in the data structures internally used by SuperLU. The routine `get_perm_c` is called to generate a column permutation vector, stored in `perm_c[]`. A good column permutation should make the L and U factors as sparse as possible. The user can supply `perm_c[]` instead of using the one provided by SuperLU. After calling the SuperLU routine `dgssv`, the B matrix is overwritten by the solution matrix X. In the end, all the dynamically allocated data structures are de-allocated by calling various utility routines.

The SuperLU package can perform more general tasks, which will be explained later.

```
#include "dsp_defs.h"
#include "util.h"

main(int argc, char *argv[])
{
    SuperMatrix A, L, U, B;
    double   *a, *rhs;
    double   s, u, p, e, r, l;
    int      *asub, *xa;
    int      *perm_r; /* row permutations from partial pivoting */
    int      *perm_c; /* column permutation vector */
    int      nrhs, info, i, m, n, nnz, permc_spec;

    /* Initialize matrix A. */
    m = n = 5;
    nnz = 12;
    if ( !(a = doubleMalloc(nnz)) ) ABORT("Malloc fails for a[].");
    if ( !(asub = intMalloc(nnz)) ) ABORT("Malloc fails for asub[].");
    if ( !(xa = intMalloc(n+1)) ) ABORT("Malloc fails for xa[].");
    s = 19.0; u = 21.0; p = 16.0; e = 5.0; r = 18.0; l = 12.0;
    a[0] = s; a[1] = l; a[2] = l; a[3] = u; a[4] = l; a[5] = l;
    a[6] = u; a[7] = p; a[8] = u; a[9] = e; a[10]= u; a[11]= r;
    asub[0] = 0; asub[1] = 1; asub[2] = 4; asub[3] = 1;
```

```
    asub[4] = 2; asub[5] = 4; asub[6] = 0; asub[7] = 2;
    asub[8] = 0; asub[9] = 3; asub[10]= 3; asub[11]= 4;
    xa[0] = 0; xa[1] = 3; xa[2] = 6; xa[3] = 8; xa[4] = 10; xa[5] = 12;

    /* Create matrix A in the format expected by SuperLU. */
    dCreate_CompCol_Matrix(&A, m, n, nnz, a, asub, xa, NC, _D, GE);

    /* Create right-hand side matrix B. */
    nrhs = 1;
    if ( !(rhs = doubleMalloc(m * nrhs)) ) ABORT("Malloc fails for rhs[].");
    for (i = 0; i < m; ++i) rhs[i] = 1.0;
    dCreate_Dense_Matrix(&B, m, nrhs, rhs, m, DN, _D, GE);

    if ( !(perm_r = intMalloc(m)) ) ABORT("Malloc fails for perm_r[].");
    if ( !(perm_c = intMalloc(n)) ) ABORT("Malloc fails for perm_c[].");

    /*
     * Get column permutation vector perm_c[], according to permc_spec:
     *   permc_spec = 0: use the natural ordering
     *   permc_spec = 1: use minimum degree ordering on structure of A'*A
     *   permc_spec = 2: use minimum degree ordering on structure of A'+A
     */
    permc_spec = 0;
    get_perm_c(permc_spec, &A, perm_c);

    dgssv(&A, perm_c, perm_r, &L, &U, &B, &info);

    printf("dgssv() returns INFO = %d\n", info);

    /* De-allocate storage */
    SUPERLU_FREE (rhs);
    SUPERLU_FREE (perm_r);
    SUPERLU_FREE (perm_c);
    Destroy_CompCol_Matrix(&A);
    Destroy_SuperMatrix_Store(&B);
    Destroy_SuperNode_Matrix(&L);
    Destroy_CompCol_Matrix(&U);
}
```

2 Matrix data structures

SuperLU uses a principal data structure `SuperMatrix` (defined in `SRC/supermatrix.h`) to represent a general matrix, sparse or dense. Figure 2 presents the specification of the `SuperMatrix` structure. The `SuperMatrix` structure contains two levels of fields. The first level defines all the properties of a matrix which are independent of how it is stored in memory. In particular, it specifies the following three orthogonal properties: storage type (`Stype`) indicates the type of the storage scheme in `*Store`; data type (`Dtype`) encodes the four precisions; mathematical type (`Mtype`) spec-

```
typedef struct {
    Stype_t Stype; /* Storage type: indicates the storage format of *Store. */
    Dtype_t Dtype; /* Data type. */
    Mtype_t Mtype; /* Mathematical type */
    int  nrow;     /* number of rows */
    int  ncol;     /* number of columns */
    void *Store;   /* pointer to the actual storage of the matrix */
} SuperMatrix;

typedef enum {
    NR,        /* row-wise, not supernodal */
    NC,        /* column-wise, not supernodal */
    SR,        /* row-wise, supernodal */
    SC,        /* column-wise, supernodal */
    NCP,       /* column-wise, not supernodal, permuted by columns
                  (After column permutation, the consecutive columns of
                   nonzeros may not be stored contiguously. */
    DN         /* Fortran style column-wise storage for dense matrix */
} Stype_t;

typedef enum {
    _S,         /* single */
    _D,         /* double */
    _C,         /* single-complex */
    _Z          /* double-complex */
} Dtype_t;

typedef enum {
    GE,        /* general */
    TRLU,      /* lower triangular, unit diagonal */
    TRUU,      /* upper triangular, unit diagonal */
    TRL,       /* lower triangular */
    TRU,       /* upper triangular */
    SYL,       /* symmetric, store lower half */
    SYU,       /* symmetric, store upper half */
    HEL,       /* Hermitian, store lower half */
    HEU        /* Hermitian, store upper half */
} Mtype_t;
```

Figure 2: SuperMatrix data structure.

ifies some mathematical properties. The second level (`*Store`) points to the actual storage used to store the matrix. We associate with each `Stype XX` a storage format called `XXformat`, such as `NPformat`, `SCformat`, etc.

The `SuperMatrix` type so defined can accommodate various types of matrix structures and appropriate operations to be applied on them, although currently SuperLU implements only a subset of this collection. Specifically, SuperLU assumes that all matrices are stored in column-major order. Matrices A, L, U, B, and X can have the following types:

	A	L	U	B	X
`Stype`	`NC` or `NCP`	`SC`	`NC`	`DN`	`DN`
`Dtype`[1]	any	any	any	any	any
`Mtype`	`GE`	`TRLU`	`TRU`	`GE`	`GE`

In what follows, we illustrate the storage schemes defined by `Stype`. Following C's convention, all array indices and locations below are 0-based.

- A may have storage type `NC` or `NCP`. The `NC` format is the same as the Harwell-Boeing sparse matrix format [5].

```
typedef struct {
    int  nnz;     /* number of nonzeros in the matrix */
    void *nzval;  /* array of nonzero values packed by column */
    int  *rowind; /* array of row indices of the nonzeros */
    int  *colptr; /* colptr[j] stores the location in nzval[] and rowind[]
                     which starts column j */
} NCformat;
```

  The `NCP` format is used when A is multiplied by a permutation matrix from the right (see Section 3). After column permutation, the consecutive columns of nonzeros may not be stored contiguously in memory. Therefore, we need two separate arrays of indices, `colbeg[]` and `colend[]`, to indicate the beginning and end of each column in `nzval[]` and `rowind[]`.

```
typedef struct {
    int  nnz;     /* number of nonzeros in the matrix */
    void *nzval;  /* array of nonzero values, packed by column */
    int  *rowind; /* array of row indices of the nonzeros */
    int  *colbeg; /* colbeg[j] stores the location in nzval[] and rowind[]
                     which starts column j */
    int  *colend; /* colend[j] stores one past the location in nzval[]
                     and rowind[] which ends column j */
} NCPformat;
```

- L is a supernodal matrix with the storage type `SC`. Due to the supernodal structure, L is in fact stored as a sparse block lower triangular matrix [4].

[1] `Dtype` can be one of `_S`, `_D`, `_C` or `_Z`.

```
typedef struct {
    int  nnz;           /* number of nonzeros in the matrix */
    int  nsuper;        /* index of the last supernode */
    void *nzval;        /* array of nonzero values packed by column */
    int  *nzval_colptr; /* nzval_colptr[j] stores the location in
                           nzval[] which starts column j */
    int  *rowind;       /* array of compressed row indices of
                           rectangular supernodes */
    int  *rowind_colptr;/* rowind_colptr[j] stores the location in
                           rowind[] which starts column j */
    int  *col_to_sup;   /* col_to_sup[j] is the supernode number to
                           which column j belongs */
    int  *sup_to_col;   /* sup_to_col[s] points to the starting column
                           of the s-th supernode */
} SCformat;
```

- Both B and X are stored as conventional two-dimensional arrays in column-major order, with the storage type DN.

```
typedef struct {
    int lda;     /* leading dimension */
    void *nzval; /* array of size lda-by-ncol to represent
                    a dense matrix */
} DNformat;
```

Figure 3 shows the data structures for the example matrices in Figure 1.

3 Permutations

Two permutation matrices are involved in the solution process. In fact, the actual factorization we perform is $P_r A P_c^T = LU$, where P_r is determined from partial pivoting (with a threshold pivoting option), and P_c is a column permutation chosen either by the user or SuperLU, usually to make the L and U factors as sparse as possible. P_r and P_c are represented by two integer vectors `perm_r[]` and `perm_c[]`, which are the permutations of the integers $(0 : m-1)$ and $(0 : n-1)$, respectively.

3.1 Ordering for sparsity

Column reordering for sparsity is completely separate from the LU factorization. The column permutation P_c should be applied before calling the factor routine `xGSTRF`. In principle, any ordering heuristic used for symmetric matrices can be applied to $A^T A$ (or $A + A^T$ if the matrix is nearly structurally symmetric) to obtain P_c. Currently, we provide the following ordering options through subroutine `get_perm_c`.

```
void get_perm_c(int ispec, SuperMatrix *A, int *perm_c);
```

`Ispec` specifies the ordering to be returned in `*perm_c`, the integer vector representing the permutation matrix P_c:

```
• A = { Stype = NC; Dtype = _D; Mtype = GE; nrow = 5; ncol = 5;
        *Store = { nnz = 12;
                   nzval = [ 19.00, 12.00, 12.00, 21.00, 12.00, 12.00, 21.00,
                             16.00, 21.00, 5.00, 21.00, 18.00 ];
                   rowind = [ 0, 1, 4, 1, 2, 4, 0, 2, 0, 3, 3, 4 ];
                   colptr = [ 0, 3, 6, 8, 10, 12 ];
                 }
      }

• U = { Stype = NC; Dtype = _D; Mtype = TRU; nrow = 5; ncol = 5;
        *Store = { nnz = 4;
                   nzval = [ 21.00, 21.00, -13.26, 7.58 ];
                   rowind = [ 0, 0, 1, 2 ];
                   colptr = [ 0, 0, 0, 1, 4, 4 ];
                 }
      }

• L = { Stype = SC; Dtype = _D; Mtype = TRLU; nrow = 5; ncol = 5;
        *Store = { nnz = 13;
                   nsuper = 2;
                   nzval = [ 19.00, 0.63, 0.63, 21.00, 0.57, 0.57, -13.26,
                             23.58, -0.24, 5.00, -0.77, 21.00, 34.20 ];
                   nzval_colptr = [ 0 3, 6, 9, 11, 13 ];
                   rowind = [ 0, 1, 4, 1, 2, 4, 3, 4 ];
                   rowind_colptr = [ 0, 3, 3, 6, 6, 8 ];
                   col_to_sup = [ 0, 1, 1, 2, 2 ];
                   sup_to_col = [ 0, 1, 3, 5 ];
                 }
      }
```

Figure 3: The data structures for a 5×5 matrix and its LU factors, as represented in the `SuperMatrix` data structure. Zero-based indexing is used.

ispec = 0: natural ordering (i.e., $P_c = I$)
= 1: MMD applied to the structure of $A^T A$
= 2: MMD applied to the structure of $A + A^T$

The MMD code is due to Joseph W.H. Liu, which implements a variant of the minimum degree ordering algorithm [7].

Alternatively, users can provide their own column permutation vector. For example, it may be an ordering suitable for the underlying physical problem. Both driver routines `xGSSV` and `xGSSVX` take `perm_c[]` as an input argument. In the future, we will augment `get_perm_c` functionality with more ordering algorithms, such as approximate minimum degree ordering on the column intersection graph of A [3].

3.2 Partial pivoting with threshold

We have included a threshold pivoting parameter $u \in [0, 1]$ to control numerical stability. The user can choose to use a row permutation obtained from a previous factorization. (The argument `*refact = 'Y'` should be passed to the factorization routine `xGSTRF`.) The pivoting subroutine `xPIVOTL` checks whether this choice of pivot satisfies the threshold; if not, it will try the diagonal element. If neither of the above satisfies the threshold, the maximum magnitude element in the column will be used as the pivot. The pseudo-code of the pivoting policy for column j is given below.

(1) compute $thresh = u\ |a_{mj}|$, where $|a_{mj}| = \max_{i \ge j} |a_{ij}|$;

(2) **if** user specifies pivot row k **and** $|a_{kj}| \ge thresh$ **and** $a_{kj} \ne 0$ **then**
        pivot row = k;
    else if $|a_{jj}| \ge thresh$ **and** $a_{jj} \ne 0$ **then**
        pivot row = j;
    else
        pivot row = m;
    endif;

Two special values of u result in the following two strategies:

- $u = 0.0$: either use user-specified pivot order if available, or else use diagonal pivot;
- $u = 1.0$: classical partial pivoting.

4 User-callable routines

The naming conventions, calling sequences and functionality of these routines mimic the corresponding LAPACK software [1]. In the routine names, such as `xGSTRF`, we use the two letters `GS` to denote *general sparse* matrices. The leading letter `x` stands for `S`, `D`, `C`, or `Z`, specifying the data type. Appendix A contains, for each individual routine, the leading comments and the complete specification of the calling sequence and arguments.

4.1 Driver routines

We provide two types of driver routines for solving systems of linear equations.

- A simple driver `xGSSV`, which solves the system $AX = B$ by factorizing A and overwriting B with the solution X.
- An expert driver `xGSSVX`, which, in addition to the above, also performs the following functions (some of them optionally):
  - solve $A^T X = B$;
  - equilibrate the system (scale A's rows and columns to have unit norm) if A is poorly scaled;
  - estimate the condition number of A, check for near-singularity, and check for pivot growth;
  - refine the solution and compute forward and backward error bounds.

These driver routines cover all the functionality of the computational routines. We expect that most users can simply use these driver routines to fulfill their tasks with no need to bother with the computational routines.

4.2 Computational routines

Users can invoke the following computational routines, instead of the driver routines, to directly control the behavior of SuperLU.

- `xGSTRF`: Factorize.

  This implements the first-time factorization, or later re-factorization with the same nonzero pattern. In re-factorizations, the code has the abilitity to use the same column permutation P_c and row permutation P_r obtained from a previous factorization. Several scalar arguments control how the LU decomposition and the numerical pivoting should be performed. `xGSTRF` can handle non-square matrices.

- `xGSTRS`: Triangular solve.

  This takes the L and U triangular factors, the row and column permutation vectors, and the right-hand side to compute a solution matrix X of $AX = B$ or $A^T X = B$.

- `xGSCON`: Estimate condition number.

  Given the matrix A and its factors L and U, this estimates the condition number in the one-norm or infinity-norm. The algorithm is due to Hager and Higham [6], and is the same as `CONDEST` in sparse Matlab.

- `xGSEQU/xLAQGS`: Equilibrate.

  `xGSEQU` first computes the row and column scalings D_r and D_c which would make each row and each column of the scaled matrix $D_r A D_c$ have equal norm. `xLAQGS` then applies them to the original matrix A if it is indeed badly scaled. The equilibrated A overwrites the original A.

- xGSRFS: Refine solution.

  Given A, its factors L and U, and an initial solution X, this does iterative refinement, using the same precision as the input data. It also computes forward and backward error bounds for the refined solution.

5 Matlab interface

In the SuperLU/MATLAB subdirectory, we have developed a set of MEX-files interface to Matlab. Right now, only the factor routine DGSTRF and the simple driver routine DGSSV are callable by invoking superlu and lusolve in Matlab, respectively. Superlu and lusolve correspond to the two Matlab built-in functions lu and \ . In Matlab, when you type

```
help superlu
```

you will find the following description about superlu's functionality and how to use it.

```
SUPERLU : Supernodal LU factorization

Executive summary:

[L,U,p] = superlu(A)          is like [L,U,P] = lu(A), but faster.
[L,U,prow,pcol] = superlu(A)  preorders the columns of A by min degree,
                                  yielding A(prow,pcol) = L*U.

Details and options:

With one input and two or three outputs, SUPERLU has the same effect as LU,
except that the pivoting permutation is returned as a vector, not a matrix:

[L,U,p] = superlu(A) returns unit lower triangular L, upper triangular U,
          and permutation vector p with A(p,:) = L*U.
[L,U] = superlu(A) returns permuted triangular L and upper triangular U
          with A = L*U.

With a second input, the columns of A are permuted before factoring:

[L,U,prow] = superlu(A,psparse) returns triangular L and U and permutation
          prow with A(prow,psparse) = L*U.
[L,U] = superlu(A,psparse) returns permuted triangular L and triangular U
          with A(:,psparse) = L*U.
Here psparse will normally be a user-supplied permutation matrix or vector
to be applied to the columns of A for sparsity.  COLMMD is one way to get
such a permutation; see below to make SUPERLU compute it automatically.
(If psparse is a permutation matrix, the matrix factored is A*psparse'.)

With a fourth output, a column permutation is computed and applied:

[L,U,prow,pcol] = superlu(A,psparse)  returns triangular L and U and
          permutations prow and pcol with A(prow,pcol) = L*U.
```

```
          Here psparse is a user-supplied column permutation for sparsity,
          and the matrix factored is A(:,psparse) (or A*psparse' if the
          input is a permutation matrix).  Output pcol is a permutation
          that first performs psparse, then postorders the etree of the
          column intersection graph of A.  The postorder does not affect
          sparsity, but makes supernodes in L consecutive.
[L,U,prow,pcol] = superlu(A,0) is the same as ... = superlu(A,I); it does
          not permute for sparsity but it does postorder the etree.
[L,U,prow,pcol] = superlu(A) is the same as ... = superlu(A,colmmd(A));
          it uses column minimum degree to permute columns for sparsity,
          then postorders the etree and factors.
```

For a description about `lusolve`'s functionality and how to use it, you can type

```
    help superlu
```

```
LUSOLVE : Solve linear systems by supernodal LU factorization.

x = lusolve(A, b) returns the solution to the linear system A*x = b,
    using a supernodal LU factorization that is faster than Matlab's
    builtin LU.  This m-file just calls a mex routine to do the work.

By default, A is preordered by column minimum degree before factorization.
Optionally, the user can supply a desired column ordering:

x = lusolve(A, b, pcol) uses pcol as a column permutation.
    It still returns x = A\b, but it factors A(:,pcol) (if pcol is a
    permutation vector) or A*Pcol (if Pcol is a permutation matrix).

x = lusolve(A, b, 0) suppresses the default minimum degree ordering;
    that is, it forces the identity permutation on columns.
```

Two M-files `trysuperlu.m` and `trylusolve.m` are written to test the correctness of `superlu` and `lusolve`. In addition to testing the residual norms, they also test the function invocations with various number of input/output arguments.

6 Memory management for L and U

In the sparse LU algorithm, the amount of space needed to hold the data structures of L and U cannot be accurately predicted prior to the factorization. The dynamically growing arrays include those for the nonzero values (`nzval`) and the compressed row indices (`rowind`) of L, and for the nonzero values (`nzval`) and the row indices (`rowind`) of U.

Two alternative memory models are presented to the user:

- system-level – based on C's dynamic allocation capability (`malloc/free`);
- user-level – based on a user-supplied `work[]` array of size `lwork` (in bytes). This is similar to Fortran-77 style handling of work space. `Work[]` is organized as a two-ended stack, one end holding the L and U data structures, the other end holding the auxiliary arrays of known size.

Except for the different ways to allocate/deallocate space, the logical view of the memory organization is the same for both schemes. Now we describe the policies in the memory module.

At the outset of the factorization, we guess there will be `FILL*nnz(A)` fills in the factors and allocate corresponding storage for the above four arrays, where `nnz(A)` is the number of nonzeros in original matrix A, and `FILL` is an integer, say 20. (The value of `FILL` can be set in an inquiry function `sp_ienv()`, see Section 7.3.) If this initial request exceeds the physical memory constraint, the `FILL` factor is repeatedly reduced, and attempts are made to allocate smaller arrays, until the initial allocation succeeds.

During the factorization, if any array size exceeds the allocated bound, we expand it as follows. We first allocate a chunk of new memory of size `EXPAND` times the old size, then copy the existing data into the new memory, and then free the old storage. The extra copying is necessary, because the factorization algorithm requires that each of the aforementioned four data structures be *contiguous* in memory. The values of `FILL` and `EXPAND` are normally set to 20 and 1.5, respectively. See `xmemory.c` for details.

After factorization, we do not garbage-collect the extra space that may have been allocated. Thus, there will be external fragmentation in the L and U data structures. The settings of `FILL` and `EXPAND` should take into account the trade-off between the number of expansions and the amount of fragmentation.

Arrays of known size, such as various column pointers and working arrays, are allocated just once. All dynamically-allocated working arrays are freed after factorization.

7 Installation

7.1 File structure

The top level SuperLU/ directory is structured as follows:

```
SuperLU/README     instructions on installation
SuperLU/CBLAS/     needed BLAS routines in C, not necessarily fast
SuperLU/EXAMPLE/   example programs
SuperLU/INSTALL/   test machine dependent parameters; this Users' Guide
SuperLU/MATLAB/    Matlab mex-file interface
SuperLU/SRC/       C source code, to be compiled into the superlu.a library
SuperLU/TESTING/   driver routines to test correctness
SuperLU/Makefile   top level Makefile that does installation and testing
SuperLU/make.inc   compiler, compile flags, library definitions and C
                   preprocessor definitions, included in all Makefiles.
```

Before installing the package, you may need to edit `SuperLU/make.inc` for your system. This make include file is referenced inside each of the `Makefiles` in the various subdirectories. As a result, there is no need to edit the `Makefiles` in the subdirectories. All information that is machine specific has been defined in `make.inc`.

Sample machine-specific `make.inc` are provided in the top-level `SuperLU/` directory for several systems, including IBM RS/6000, DEC Alpha, SunOS 4.x, SunOS 5.x (Solaris), HP-PA and SGI Iris 4.x. When you have selected the machine on which you wish to install SuperLU, you may copy the appropriate sample include file (if one is present) into `make.inc`. For example, if you wish to run SuperLU on an IBM RS/6000, you can do:

```
cp make.rs6k make.inc
```

For systems other than those listed above, slight modifications to the `make.inc` file will need to be made. In particular, the following three items should be examined:

1. The BLAS library.
   If there is a BLAS library available on your machine, you may define the following in `make.inc`:

   ```
   BLASDEF = -DUSE_VENDOR_BLAS
   BLASLIB = <BLAS library you wish to link with>
   ```

   The `CBLAS/` subdirectory contains the part of the C BLAS needed by the SuperLU package. However, these codes are intended for use only if there is no faster implementation of the BLAS already available on your machine. In this case, you should do the following:

   1) In `make.inc`, define:
      ```
      BLASLIB = ../blas$(PLAT).a
      ```
   2) In the SuperLU/ directory, type:
      ```
      make blaslib
      ```
      to make the BLAS library from the routines in the `CBLAS/` subdirectory.

2. C preprocessor definition `CDEFS`.
   In the header file `SRC/Cnames.h`, we use macros to determine how C routines should be named so that they are callable by Fortran.[2] The possible options for `CDEFS` are:

   - `-DAdd_`: Fortran expects a C routine to have an underscore postfixed to the name;
   - `-DNoChange`: Fortran expects a C routine name to be identical to that compiled by C;
   - `-DUpCase`: Fortran expects a C routine name to be all uppercase.

3. The Matlab MEX-file interface.
   The `MATLAB/` subdirectory includes Matlab C MEX-files, so that our factor and solve routines can be called as alternatives to those built into Matlab. In the file `SuperLU/make.inc`, define MATLAB to be the directory in which Matlab is installed on your system, for example:

   ```
   MATLAB = /usr/local/matlab
   ```

   At the SuperLU/ directory, type:

   ```
   make matlabmex
   ```

   to build the MEX-file interface. After you have built the interface, you may go to the `MATLAB/` subdirectory to test the correctness by typing (in Matlab):

   ```
   trysuperlu
   trylusolve
   ```

A `Makefile` is provided in each subdirectory. The installation can be done completely automatically by simply typing `make` at the top level.

[2]Some vendor-supplied BLAS libraries do not have C interface. So the re-naming is needed in order for the SuperLU BLAS calls (in C) to interface with the Fortran-style BLAS.

Matrix type	Description
0	sparse matrix g10
1	diagonal
2	upper triangular
3	lower triangular
4	random, $\kappa = 2$
5	first column zero
6	last column zero
7	last $n/2$ columns zero
8	random, $\kappa = \sqrt{0.1/\varepsilon}$
9	random, $\kappa = 0.1/\varepsilon$
10	scaled near underflow
11	scaled near overflow

Table 1: Properties of the test matrices. ε is the machine epsilon and κ is the condition number of matrix A. Matrix types with one or more columns set to zero are used to test the error return codes.

Test ratio	Routines
$\|\|LU - A\|\|/(n\|\|A\|\|\varepsilon)$	xGSTRF
$\|\|b - Ax\|\|/(\|\|A\|\|\ \|\|x\|\|\varepsilon)$	xGSSV, xGSSVX
$\|\|x - x^*\|\|/(\|\|x^*\|\|\kappa\varepsilon)$	xGSSVX
$\|\|x - x^*\|\|/(\|\|x^*\|\|\ \texttt{FERR})$	xGSSVX
$\texttt{BERR}/\varepsilon$	xGSSVX

Table 2: Types of tests. x^* is the exact solution, FERR is the error bound, and BERR is the backward error.

7.2 Testing

The test programs in SuperLU/INSTALL subdirectory test two routines:

- SLAMCH/DLAMCH determines properties of the floating-point arithmetic at run-time (both single and double precision), such as the machine epsilon, underflow threshold, overflow threshold, and related parameters;

- SuperLU_timer_() returns the time in seconds used by the process. This function may need to be modified to run on your machine.

The test programs in the SuperLU/TESTING subdirectory are designed to test all the functions of the driver routines, especially the expert drivers. The Unix shell script files xtest.csh are used to invoke tests with varying parameter settings. The input matrices include an actual sparse matrix SuperLU/EXAMPLE/g10 of dimension 100×100,[3] and numerous matrices with special properties from the LAPACK test suite. Table 1 describes the properties of the test matrices.

For each command line option specified in xtest.csh, the test program xDRIVE reads in or generates an appropriate matrix, calls the driver routines, and computes a number of test ratios to verify that each operation has performed correctly. If the test ratio is smaller than a preset threshold, the operation is considered to be correct. Each test matrix is subject to the tests listed in Table 2.

Let r be the residual $r = b - Ax$, and let m_i be the number of nonzeros in row i of A. Then BERR and FERR are calculated by:

$$\texttt{BERR} = \max_i \frac{|r|_i}{(|A|\ |x| + |b|)_i}\ .$$

[3]Matrix g10 is first generated with the structure of the 10-by-10 five-point grid, and random numerical values. The columns are then permutated by COLMMD ordering from Matlab.

$$\texttt{FERR} = \frac{||\ |A^{-1}|\ f\ ||_\infty}{||x||_\infty}\ .$$

Here, f is a nonnegative vector whose components are computed as $f_i = |r|_i + m_i\,\varepsilon\,(|A|\ |x| + |b|)_i$, and the norm in the numerator is estimated using the same subroutine used for estimating the condition number. For further details on error analysis and error bounds estimation, see [1, Chapter 4] and [2].

7.3 Performance-tuning parameters

SuperLU chooses such machine-dependent parameters as block size by calling an inquiry function `sp_ienv()`, which may be set to return different values on different machines. The declaration of this function is

```
int sp_ienv(int ispec);
```

`Ispec` specifies the parameter to be returned, (See reference [4] for their definitions.)

ispec = 1: the panel size (w)
= 2: the relaxation parameter to control supernode amalgamation ($relax$)
= 3: the maximum allowable size for a supernode ($maxsuper$)
= 4: the minimum row dimension for 2-D blocking to be used ($rowblk$)
= 5: the minimum column dimension for 2-D blocking to be used ($colblk$)
= 6: the estimated fills factor for L and U, compared with A;

Users are encouraged to modify this subroutine to set the tuning parameters for their own local environment. The optimal values depend mainly on the cache size and the BLAS speed. If your system has a very small cache, or if you want to efficiently utilize the closest cache in a multilevel cache organization, you should pay special attention to these parameter settings. In our technical paper [4], we described a detailed methodology for setting these parameters for high performance.

The $relax$ parameter is usually set between 4 and 8. The other parameter values which give good performance on several machines are listed in Table 3. In a supernode-panel update, if the updating supernode is too large to fit in cache, then a 2-D block partitioning of the supernode is used, in which $rowblk$ and $colblk$ determine that a block of size $rowblk \times colblk$ is used to update current panel.

If $colblk$ is set greater than $maxsup$, then the program will never use 2-D blocking. For example, for the Cray J90 (which does not have cache), $w = 1$ and 1-D blocking give good performance; more levels of blocking only increase overhead.

7.4 Error handling

A macro `ABORT` is defined in `SRC/util.h` to handle unrecoverable errors that occur in the middle of the computation, such as `malloc` failure. The default action of `ABORT` is to call

```
superlu_abort_and_exit(char *msg)
```

which prints an error message, the line number and the file name at which the error occurs, and calls `exit` function to terminate the program.

If this type of termination is not appropriate in some environment, users can define their own abort function. When compiling the SuperLU library, users choose the C preprocessor definition

```
-DUSER_ABORT = my_abort
```

At the same time, users supply the following `my_abort` function

Machine	On-chip Cache	External Cache	w	$maxsup$	$rowblk$	$colblk$
RS/6000-590	256 KB	–	8	100	200	40
MIPS R8000	16 KB	4 MB	20	100	800	100
Alpha 21064	8 KB	512 KB	8	100	400	40
Alpha 21164	8 KB-L1 96 KB-L2	4 MB	16	50	100	40
Sparc 20	16 KB	1 MB	8	100	400	50
UltraSparc-I	16 KB	512 KB	8	100	400	40
Cray J90	–	–	1	100	1000	101

Table 3: Typical blocking parameter values for several machines.

```
my_abort(char *msg)
```

which overrides the behavior of `superlu_abort_and_exit`.

8 Statistics

SuperLU internally records some performance statistics, such as floating-point operation counts and the time taken by factorization. A variable `SuperLUStat` is declared with the following type.

```
typedef struct {
    int     *panel_histo; /* histogram of panel size distribution */
    double  *utime;       /* time spent in various phases */
    float   *ops;         /* float-point operation count in various phases */
} SuperLUStat_t;
```

In the beginning of both driver routines `xGSSV` and `xGSSVX`, subroutine `StatInit` is called to allocate storage and perform initialization for the fields `panel_histo`, `utime`, and `ops`. In the end of the driver routines, subroutine `StatFree` is called to deallocate storage of the above statistics fields. After deallocation, the statistics are no longer accessible. Therefore, users should extract the information they need before calling `StatFree`.

An inquiry function `xQuerySpace` is provided to compute memory usage statistics. This routine should be called after the LU factorization. It calculates the storage requirement based on the size of the L and U data structures and working arrays.

9 Example programs

In the `SuperLU/EXAMPLE/` subdirectory, we present a few sample programs, such as `xLINSOL` and `xLINSOLX`, to illustrate the complete calling sequences used to solve systems of equations. These include how to set up the matrix structures, how to obtain a fill-reducing ordering, and how to call driver routines. A `Makefile` is provided to generate the executables. A `README` file in this directory shows how to run these examples.

Based on these sample programs, we now illustrate how we may use SuperLU in some other ways.

```
main()
{
    /* Declare variables */
    SuperMatrix A;  /* original matrix */
    SuperMatrix AC; /* A postmultiplied by a permutation matrix Q */
    char   refact[1];
    ...... /* declarations of other variables */

    /* Initialization */
    {
        StatInit(panel_size, relax);
        ......
    }

    /* First-time factorization */
    *refact = 'N';

    /* Obtain and apply column permutation */
    get_perm_c(1, &A, perm_c);
    sp_preorder(refact, &A, perm_c, etree, &AC);

    /* Factorization */
    dgstrf(refact, &AC, 1.0, 0.0, relax, panel_size,
           etree, NULL, 0, perm_r, &L, &U, &info);
    ...... /* solve first system */

    /* Subsequent factorizations */
    *refact = 'Y';

    for ( i = 1; i <= niter; ++i ) {
        dgstrf(refact, &AC, 1.0, 0.0, relax, panel_size,
               etree, NULL, 0, perm_r, &L, &U, &info);
        /* Numerical values of matrix AC may change across iterations.
           The factors L and U are overwritten in each iteration. */
        {
               ...... /* solve later system */
        }
    }

    StatFree();

}
```

Figure 4: Code segment to perform repeated factorizations.

9.1 Repeated factorizations

In many iterative processes, matrices with the same sparsity pattern but different numerical values must be factored repeatedly. Thus, computing a fill-reducing ordering and performing column permutation are needed only once. In addition, the memory for L and U can be allocated only once, and re-used in the subsequent factorizations. If there is not enough space for L and U from the previous factorization (due to different pivoting), the factor routines `xGSTRF` automatically expand memory as needed. Figure 4 shows the code segment for this purpose.

9.2 Calling from Fortran

General rules for mixing Fortran and C programs are as follows.

- Arguments in C are passed by value, while in Fortran are passed by reference. So we always pass the address (as a pointer) in the C calling routine. (You cannot make a call with numbers directly in the parameters.)
- Fortran uses 1-based array addressing, while C uses 0-based. Therefore, the row indices (`rowind`) and integer pointers to arrays (`colptr`) should be adjusted before they are passed into a C routine.

Because of the above language differences, in order to embed SuperLU in a Fortran environment, users are required to supply "bridge" routines (in C) for all the SuperLU subroutines that will be called from Fortran programs. Figure 5 is an example showing how a bridge program should be written. See the files `f77_main.f` and `c_bridge_dgssv.c` for complete descriptions.

In the future, we will provide complete Fortran interfaces to the user-callable routines in the SuperLU library.

10 Acknowledgement

We would like to thank Jinqchong Teo for helping generate the code to work with four floating-point data types. We also thank Sivan Toledo for suggestions on improving the routines' interfaces.

```
Fortran program (f77_main.f)
~~~~~~~~~~~~~~~

      program f77_main
      integer maxn, maxnz
      parameter ( maxn = 10000, maxnz = 100000 )
      integer rowind(maxnz), colptr(maxn)
      real*8  values(maxnz), b(maxn)
      ......

      call c_bridge_dgssv( n, nnz, nrhs, values, rowind, colptr, b, ldb, info )
      ......

      stop
      end


The bridge program in C (c_bridge_dgssv.c)
~~~~~~~~~~~~~~~~~~~~~~

    int c_bridge_dgssv(int *n, int *nnz, int *nrhs, double *values, int *rowind,
                       int *colptr, double *b, int *ldb, int *info)
    {
        SuperMatrix A, B, L, U;
        int *perm_c, *perm_r;
        ......

        /* Adjust to 0-based indexing */
        for (i = 0; i < *nnz; ++i) --rowind[i];
        for (i = 0; i <= *n; ++i) --colptr[i];

        /* Construct Matrix structures A and B */
        dCreate_CompCol_Matrix(&A, *n, *n, *nnz, values, rowind, colptr,
                               NC, _D, GE);
        dCreate_Dense_Matrix(&B, *n, *nrhs, b, *ldb, DN, _D, GE);
        ......

        /* B is overwritten by the solution vector */
        dgssv(&A, perm_c, perm_r, &L, &U, &B, info);
        ......
    }
```

Figure 5: Interface with Fortran

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Release 2.0.* SIAM, Philadelphia, 1995. 324 pages.

[2] M. Arioli, J. W. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Appl.*, 10(2):165–190, April 1989.

[3] Timothy A. Davis, John R. Gilbert, Esmond Ng, and Barry Peyton. Approximate minimum degree ordering for unsymmetric matrices. Talk presented at XIII Householder Symposium on Numerical Algebra, June 1996. Journal version in preparation.

[4] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W.H. Liu. A supernodal approach to sparse partial pivoting. Technical Report UCB//CSD-95-883, Computer Science Division, U.C. Berkeley, July 1995. (Xerox PARC report CSL-95-03, LAPACK Working Note #103).

[5] I.S Duff, R.G Grimes, and J.G Lewis. Users' guide for the harwell-boeing sparse matrix collection (release 1). Technical Report RAL-92-086, Rutherford Appleton Laboratory, December 1992.

[6] N. J. Higham. Algorithm 674: FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation. *ACM Trans. Math. Soft.*, 14:381–396, 1988.

[7] Joseph W.H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11:141–153, 1985.

A Specifications of routines

A.1 SGSEQU

```
void
sgsequ(SuperMatrix *A, float *r, float *c, float *rowcnd,
       float *colcnd, float *amax, int *info)

    Purpose
    =======

    SGSEQU computes row and column scalings intended to equilibrate an
    M-by-N sparse matrix A and reduce its condition number. R returns the
    row scale factors and C the column scale factors, chosen to try to make
    the largest element in each row and column of the matrix B with
    elements B(i,j)=R(i)*A(i,j)*C(j) have absolute value 1.

    R(i) and C(j) are restricted to be between SMLNUM = smallest safe
    number and BIGNUM = largest safe number.  Use of these scaling
    factors is not guaranteed to reduce the condition number of A but
    works well in practice.

    See supermatrix.h for the definition of 'SuperMatrix' structure.

    Arguments
    =========

    A       (input) SuperMatrix*
            The matrix of dimension (A->nrow, A->ncol) whose equilibration
            factors are to be computed. The type of A can be:
            Stype = NC; Dtype = _S; Mtype = GE.

    R       (output) float*, dimension (A->nrow)
            If INFO = 0 or INFO > M, R contains the row scale factors
            for A.

    C       (output) float*, dimension (A->ncol)
            If INFO = 0,  C contains the column scale factors for A.

    rowcnd  (output) float*
            If INFO = 0 or INFO > M, ROWCND contains the ratio of the
            smallest R(i) to the largest R(i).  If ROWCND >= 0.1 and
            AMAX is neither too large nor too small, it is not worth
            scaling by R.

    colcnd  (output) float*
            If INFO = 0, COLCND contains the ratio of the smallest
            C(i) to the largest C(i).  If COLCND >= 0.1, it is not
```

```
        worth scaling by C.

amax    (output) float*
        Absolute value of largest matrix element.  If AMAX is very
        close to overflow or very close to underflow, the matrix
        should be scaled.

info    (output) int*
        = 0:  successful exit
        < 0:  if INFO = -i, the i-th argument had an illegal value
        > 0:  if INFO = i,  and i is
              <= M:  the i-th row of A is exactly zero
              >  M:  the (i-M)-th column of A is exactly zero
```

A.2 SGSCON

```
void
sgscon(char *norm, SuperMatrix *L, SuperMatrix *U,
       float anorm, float *rcond, int *info)

    Purpose
    =======

    SGSCON estimates the reciprocal of the condition number of a general
    real matrix A, in either the 1-norm or the infinity-norm, using
    the LU factorization computed by SGETRF.

    An estimate is obtained for norm(inv(A)), and the reciprocal of the
    condition number is computed as
       RCOND = 1 / ( norm(A) * norm(inv(A)) ).

    See supermatrix.h for the definition of 'SuperMatrix' structure.

    Arguments
    =========

    NORM    (input) char*
            Specifies whether the 1-norm condition number or the
            infinity-norm condition number is required:
            = '1' or 'O':  1-norm;
            = 'I':         Infinity-norm.

    A       (input) SuperMatrix*
            The original matrix, or equilibrated matrix.

    L       (input) SuperMatrix*
            The factor L from the factorization Pr*A*Pc=L*U as computed by
```

```
        SGSTRF. Use compressed row subscripts storage for supernodes,
        i.e., L has types: Stype = SC, Dtype = _S, Mtype = TL.

U       (input) SuperMatrix*
        The factor U from the factorization Pr*A*Pc=L*U as computed by
        SGSTRF. Use column-wise storage scheme, i.e., U has types:
        Stype = NC, Dtype = _S, Mtype = TU.

anorm   (input) float
        If NORM = '1' or 'O', the 1-norm of the original matrix A.
        If NORM = 'I', the infinity-norm of the original matrix A.

rcond   (output) float*
        The reciprocal of the condition number of the matrix A,
        computed as RCOND = 1/(norm(A) * norm(inv(A))).

info    (output) int*
        = 0:  successful exit
        < 0:  if INFO = -i, the i-th argument had an illegal value
```

A.3 SGSRFS

```
void
sgsrfs(char *trans, SuperMatrix *A, SuperMatrix *L, SuperMatrix *U,
       int *perm_r, int *perm_c, char *equed, float *R, float *C,
       SuperMatrix *B, SuperMatrix *X, float *ferr, float *berr, int *info)

    Purpose
    =======

    SGSRFS improves the computed solution to a system of linear
    equations and provides error bounds and backward error estimates for
    the solution.

    See supermatrix.h for the definition of 'SuperMatrix' structure.

    Arguments
    =========

    trans   (input) char*
            Specifies the form of the system of equations:
            = 'N':  A * X = B     (No transpose)
            = 'T':  A**T * X = B  (Transpose)
            = 'C':  A**H * X = B  (Conjugate transpose = Transpose)

    A       (input) SuperMatrix*
```

```
        The original matrix A in the system, or the scaled A if
        equilibration was done.

L       (input) SuperMatrix*
        The factor L from the factorization Pr*A*Pc=L*U. Use
        compressed row subscripts storage for supernodes,
        i.e., L has types: Stype = SC, Dtype = _S, Mtype = TL.

U       (input) SuperMatrix*
        The factor U from the factorization Pr*A*Pc=L*U as computed by
        SGSTRF. Use column-wise storage scheme,
        i.e., U has types: Stype = NC, Dtype = _S, Mtype = TU.

perm_r  (input) int*, dimension (A->nrow)
        Row permutation vector, which defines the permutation matrix Pr;
        perm_r[i] = j means row i of A is in position j in Pr*A.

perm_c  (input) int*, dimension (A->ncol)
        Column permutation vector, which defines the
        permutation matrix Pc; perm_c[i] = j means column i of A is
        in position j in A*Pc.

equed   (input) Specifies the form of equilibration that was done.
        = 'N': No equilibration.
        = 'R': Row equilibration, i.e., A was premultiplied by diag(R).
        = 'C': Column equilibration, i.e., A was postmultiplied by
               diag(C).
        = 'B': Both row and column equilibration, i.e., A was replaced
               by diag(R)*A*diag(C).

R       (input) float*, dimension (A->nrow)
        The row scale factors for A.
        If equed = 'R' or 'B', A is premultiplied by diag(R).
        If equed = 'N' or 'C', R is not accessed.

C       (input) float*, dimension (A->ncol)
        The column scale factors for A.
        If equed = 'C' or 'B', A is postmultiplied by diag(C).
        If equed = 'N' or 'R', C is not accessed.

B       (input) SuperMatrix*
        B has types: Stype = DN, Dtype = _S, Mtype = GE.
        The right hand side matrix B.

X       (input/output) SuperMatrix*
        X has types: Stype = DN, Dtype = _S, Mtype = GE.
        On entry, the solution matrix X, as computed by sgstrs().
```

```
        On exit, the improved solution matrix X.

FERR    (output) float*, dimension (B->ncol)
        The estimated forward error bound for each solution vector
        X(j) (the j-th column of the solution matrix X).
        If XTRUE is the true solution corresponding to X(j), FERR(j)
        is an estimated upper bound for the magnitude of the largest
        element in (X(j) - XTRUE) divided by the magnitude of the
        largest element in X(j).  The estimate is as reliable as
        the estimate for RCOND, and is almost always a slight
        overestimate of the true error.

BERR    (output) float*, dimension (B->ncol)
        The componentwise relative backward error of each solution
        vector X(j) (i.e., the smallest relative change in
        any element of A or B that makes X(j) an exact solution).

info    (output) int*
        = 0:  successful exit
         < 0:  if INFO = -i, the i-th argument had an illegal value
```

A.4 SGSSV

```
void
sgssv(SuperMatrix *A, int *perm_c, int *perm_r, SuperMatrix *L,
      SuperMatrix *U, SuperMatrix *B, int *info )

    Purpose
    =======

    SGSSV solves the system of linear equations A*X=B, using the
    LU factorization from SGSTRF. It performs the following steps:

      1. Permute columns of A, forming A*Pc, where Pc is a permutation matrix.
         For more details of this step, see sp_preorder.c.

      2. Factor A as Pr*A*Pc=L*U using LU decomposition with Pr determined
         by partial pivoting.

      3. Solve the system of equations A*X=B using the factored form of A.

      See supermatrix.h for the definition of 'SuperMatrix' structure.

    Arguments
    =========
```

```
A       (input) SuperMatrix*
        Matrix A in A*X=B, of dimension (A->nrow, A->ncol). The number
        of linear equations is A->nrow. Currently, the type of A can be:
        Stype = NC; Dtype = _S; Mtype = GE. In the future, more
        general A can be handled.

perm_c  (input/output) int*, dimension (A->ncol)
        Column permutation vector, which defines the
        permutation matrix Pc; perm_c[i] = j means column i of A is
        in position j in A*Pc.
        On exit, perm_c may be overwritten by the product of the input
        perm_c and a permutation that postorders the elimination tree
        of Pc'*A'*A*Pc; perm_c is not changed if the elimination tree
        is already in postorder.

perm_r  (output) int*, dimension (A->nrow)
        Row permutation vector, which defines the
        permutation matrix Pr, and is determined by partial pivoting;
        perm_r[i] = j means row i of A is in position j in Pr*A.

L       (output) SuperMatrix*
        The factor L from the factorization Pr*A*Pc=L*U. Use
        compressed row subscripts storage for supernodes, i.e., L
        has types: Stype = SC, Dtype = _S, Mtype = TL.

U       (output) SuperMatrix*
        The factor U from the factorization Pr*A*Pc=L*U. Use column-wise
        storage scheme, i.e., U has types: Stype = NC, Dtype = _S,
        Mtype = TU.

B       (input/output) SuperMatrix*
        B has types: Stype = DN, Dtype = _S, Mtype = GE.
        On entry, the right hand side matrix.
        On exit, the solution matrix if info = 0;

info    (output) int*
        = 0: successful exit
        > 0: if info = i, and i is
            <= A->ncol: U(i,i) is exactly zero. The factorization has
               been completed, but the factor U is exactly singular,
               so the solution could not be computed.
            > A->ncol: number of bytes allocated when memory allocation
               failure occurred, plus A->ncol.
```

A.5 SGSSVX

```
void
sgssvx(char *fact, char *trans, char *refact, SuperMatrix *A,
       factor_param_t *factor_params, int *perm_c, int *perm_r, int *etree,
       char *equed, float *R, float *C, SuperMatrix *L, SuperMatrix *U,
       void *work, int lwork, SuperMatrix *B, SuperMatrix *X,
       float *recip_pivot_growth, float *rcond, float *ferr, float *berr,
       mem_usage_t *mem_usage, int *info )

    Purpose
    =======

    SGSSVX solves the system of linear equations A*X=B or A'*X=B, using
    the LU factorization from SGSTRF. Error bounds on the solution and
    a condition estimate are also provided. It performs the following steps:

     1. If fact = 'E', scaling factors are computed to equilibrate the system:
           trans = 'N':  diag(R)*A*diag(C)     *inv(diag(C))*X = diag(R)*B
           trans = 'T': (diag(R)*A*diag(C))**T *inv(diag(R))*X = diag(C)*B
           trans = 'C': (diag(R)*A*diag(C))**H *inv(diag(R))*X = diag(C)*B
        Whether or not the system will be equilibrated depends on the
        scaling of the matrix A, but if equilibration is used, A is
        overwritten by diag(R)*A*diag(C) and B by diag(R)*B (if trans='N')
        or diag(C)*B (if trans = 'T' or 'C').

     2. Permute columns of A, forming A*Pc, where Pc is a permutation matrix
        that usually preserves sparisity.
        For more details of this step, see sp_preorder.c.

     3. If fact = 'N' or 'E', the LU decomposition is used to factor the
        matrix A (after equilibration if fact = 'E') as Pr*A*Pc = L*U,
        with Pr determined by partial pivoting.

     4. Compute the reciprocal pivot growth factor.

     5. If some U(i,i) = 0, so that U is exactly singular, then the routine
        returns with info = i. Otherwise, the factored form of A is used to
        estimate the condition number of the matrix A. If the reciprocal of
        the condition number is less than machine precision, info = A->ncol+1
        is returned as a warning, but the routine still goes on to solve
        for X and computes error bounds as described below.

     6. The system of equations is solved for X using the factored form of A.

     7. Iterative refinement is applied to improve the computed solution matrix
        and calculate error bounds and backward error estimates for it.

     8. If equilibration was used, the matrix X is premultiplied by
```

```
    diag(C) (if trans = 'N') or diag(R) (if trans = 'T' or 'C') so
    that it solves the original system before equilibration.

 See supermatrix.h for the definition of 'SuperMatrix' structure.

Arguments
=========

fact    (input) char*
        Specifies whether or not the factored form of the matrix
        A is supplied on entry, and if not, whether the matrix A should
        be equilibrated before it is factored.
        = 'F': On entry, L, U, perm_r and perm_c contain the factored
               form of A. If equed is not 'N', the matrix A has been
               equilibrated with scaling factors R and C.
               A, L, U, perm_r are not modified.
        = 'N': The matrix A will be factored, and the factors will be
               stored in L and U.
        = 'E': The matrix A will be equilibrated if necessary, then
               factored into L and U.

trans   (input) char*
        Specifies the form of the system of equations:
        = 'N': A * X = B        (No transpose)
        = 'T': A**T * X = B     (Transpose)
        = 'C': A**H * X = B     (Transpose)

refact  (input) char*
        Specifies whether we want to re-factor the matrix.
        = 'N': Factor the matrix A.
        = 'Y': Matrix A was factored before, now we want to re-factor
               matrix A with perm_r and etree as inputs. Use
               the same storage for the L\U factors previously allocated,
               expand it if necessary. User should insure to use the same
               memory model.
        If fact = 'F', then refact is not accessed.

A       (input) SuperMatrix*
        Matrix A in A*X=B, of dimension (A->nrow, A->ncol). The number of
        the linear equations is A->nrow. Currently, the type of A can be:
        Stype = NC; Dtype = _S; Mtype = GE.
        In the future, more general A may be handled.

factor_params (input) factor_param_t*
        The structure defines the input scalar parameters, consisting of
        the following fields. If factor_params = NULL, the default
        values are used for all the fields; otherwise, the values
```

```
         are given by the user.
         - panel_size (int): Panel size. A panel consists of at most
             panel_size consecutive columns. If panel_size = -1, use
             default value 8.
         - relax (int): To control degree of relaxing supernodes. If the
             number of nodes (columns) in a subtree of the elimination
             tree is less than relax, this subtree is considered as one
             supernode, regardless of the row structures of those columns.
             If relax = -1, use default value 8.
         - diag_pivot_thresh (float): Diagonal pivoting threshold.
             At step j of the Gaussian elimination, if
                 abs(A_jj) >= diag_pivot_thresh * (max_(i>=j) abs(A_ij)),
             then use A_jj as pivot. 0 <= diag_pivot_thresh <= 1.
             If diag_pivot_thresh = -1, use default value 1.0,
             which corresponds to standard partial pivoting.
         - drop_tol (double): Drop tolerance threshold. (NOT IMPLEMENTED)
             At step j of the Gaussian elimination, if
                 abs(A_ij)/(max_i abs(A_ij)) < drop_tol,
             then drop entry A_ij. 0 <= drop_tol <= 1.
             If drop_tol = -1, use default value 0.0, which corresponds to
             standard Gaussian elimination.

perm_c   (input/output) int*, dimension (A->ncol)
         Column permutation vector, which defines the
         permutation matrix Pc; perm_c[i] = j means column i of A is
         in position j in A*Pc.
         On exit, perm_c may be overwritten by the product of the input
         perm_c and a permutation that postorders the elimination tree
         of Pc'*A'*A*Pc; perm_c is not changed if the elimination tree
         is already in postorder.

perm_r   (input/output) int*, dimension (A->nrow)
         Row permutation vector, which defines the permutation matrix Pr;
         perm_r[i] = j means row i of A is in position j in Pr*A.
         If refact is not 'Y', perm_r is output argument;
         If refact = 'Y', the pivoting routine will try to use the input
         perm_r, unless a certain threshold criterion is violated.
         In that case, perm_r is overwritten by a new permutation
         determined by partial pivoting or diagonal threshold pivoting.

etree    (input/output) int*,  dimension (A->ncol)
         Elimination tree of Pc'*A'*A*Pc.
         If fact is not 'F' and refact = 'Y', etree is an input argument,
         otherwise it is an output argument.
         Note: etree is a vector of parent pointers for a forest whose
         vertices are the integers 0 to A->ncol-1; etree[root]==A->ncol.
```

```
equed   (input/output) char*
        Specifies the form of equilibration that was done.
        = 'N': No equilibration.
        = 'R': Row equilibration, i.e., A was premultiplied by diag(R).
        = 'C': Column equilibration, i.e., A was postmultiplied by diag(C).
        = 'B': Both row and column equilibration, i.e., A was replaced
               by diag(R)*A*diag(C).
        If fact = 'F', equed is an input argument, otherwise it is
        an output argument.

R       (input/output) float*, dimension (A->nrow)
        The row scale factors for A.
        If equed = 'R' or 'B', A is multiplied on the left by diag(R).
        If equed = 'N' or 'C', R is not accessed.
        If fact = 'F', R is an input argument; otherwise, R is output.
        If fact = 'F' and equed = 'R' or 'B', each element of R must
           be positive.

C       (input/output) float*, dimension (A->ncol)
        The column scale factors for A.
        If equed = 'C' or 'B', A is multiplied on the right by diag(C).
        If equed = 'N' or 'R', C is not accessed.
        If fact = 'F', C is an input argument; otherwise, C is output.
        If fact = 'F' and equed = 'C' or 'B', each element of C must
           be positive.

L       (output) SuperMatrix*
        The factor L from the factorization Pr*A*Pc=L*U. Use
        compressed row subscripts storage for supernodes, i.e., L
        has types: Stype = SC, Dtype = _S, Mtype = TL.

U       (output) SuperMatrix*
        The factor U from the factorization Pr*A*Pc=L*U. Use column-wise
        storage scheme, i.e., U has types: Stype = NC, Dtype = _S,
        Mtype = TU.

work    (workspace/output) void*, size (lwork) (in bytes)
        User supplied workspace, should be large enough
        to hold data structures for factors L and U.
        On exit, if fact is not 'F', L and U point to this array.

lwork   (input) int
        Specifies the size of work array in bytes.
        = 0:  allocate space internally by system malloc;
        > 0:  use user-supplied work array of length lwork in bytes,
              returns error if space runs out.
        = -1: the routine guesses the amount of space needed without
```

```
              performing the factorization, and returns it in
              mem_usage->total_needed; no other side effects.

        See argument 'mem_usage' for memory usage statistics.

B       (input/output) SuperMatrix*
        B has types: Stype = DN, Dtype = _S, Mtype = GE.
        On entry, the right hand side matrix.
        On exit,
           if equed = 'N', B is not modified;
           if trans = 'N' and equed = 'R' or 'B', B is overwritten by
              diag(R)*B;
           if trans = 'T' or 'C' and equed = 'C' of 'B', B is
              overwritten by diag(C)*B.

X       (output) SuperMatrix*
        X has types: Stype = DN, Dtype = _S, Mtype = GE.
        If info = 0 or info = A->ncol+1, X contains the solution matrix
        to the original system of equations. Note that A and B are modified
        on exit if equed is not 'N', and the solution to the equilibrated
        system is inv(diag(C))*X if trans = 'N' and equed = 'C' or 'B',
        or inv(diag(R))*X if trans = 'T' or 'C' and equed = 'R' or 'B'.

recip_pivot_growth (output) float*
        The reciprocal pivot growth factor max_j( norm(A_j)/norm(U_j) ).
        The infinity norm is used. If recip_pivot_growth is much less
        than 1, the stability of the LU factorization could be poor.

rcond   (output) float*
        The estimate of the reciprocal condition number of the matrix A
        after equilibration (if done). If rcond is less than the machine
        precision (in particular, if rcond = 0), the matrix is singular
        to working precision. This condition is indicated by a return
        code of info > 0.

FERR    (output) float*, dimension (B->ncol)
        The estimated forward error bound for each solution vector
        X(j) (the j-th column of the solution matrix X).
        If XTRUE is the true solution corresponding to X(j), FERR(j)
        is an estimated upper bound for the magnitude of the largest
        element in (X(j) - XTRUE) divided by the magnitude of the
        largest element in X(j).  The estimate is as reliable as
        the estimate for RCOND, and is almost always a slight
        overestimate of the true error.

BERR    (output) float*, dimension (B->ncol)
        The componentwise relative backward error of each solution
```

```
        vector X(j) (i.e., the smallest relative change in
        any element of A or B that makes X(j) an exact solution).

mem_usage (output) mem_usage_t*
        Record the memory usage statistics, consisting of following fields:
        - for_lu (float)
          The amount of space used in bytes for L\U data structures.
        - total_needed (float)
          The amount of space needed in bytes to perform factorization.
        - expansions (int)
          The number of memory expansions during the LU factorization.

info    (output) int*
        = 0: successful exit
        < 0: if info = -i, the i-th argument had an illegal value
        > 0: if info = i, and i is
            <= A->ncol: U(i,i) is exactly zero. The factorization has
                  been completed, but the factor U is exactly
                  singular, so the solution and error bounds
                  could not be computed.
            = A->ncol+1: U is nonsingular, but RCOND is less than machine
                  precision, meaning that the matrix is singular to
                  working precision. Nevertheless, the solution and
                  error bounds are computed because there are a number
                  of situations where the computed solution can be more
                  accurate than the value of RCOND would suggest.
            > A->ncol+1: number of bytes allocated when memory allocation
                  failure occurred, plus A->ncol.
```

A.6 SGSTRF

```
void
sgstrf(char *refact, SuperMatrix *A, float diag_pivot_thresh, float drop_tol,
       int relax, int panel_size, int *etree, void *work, int lwork,
       int *perm_r, int *perm_c, SuperMatrix *L, SuperMatrix *U, int *info)

    Purpose
    =======

    SGSTRF computes an LU factorization of a general sparse m-by-n
    matrix A using partial pivoting with row interchanges.
    The factorization has the form
        Pr * A = L * U
    where Pr is a row permutation matrix, L is lower triangular with unit
    diagonal elements (lower trapezoidal if A->nrow > A->ncol), and U is upper
    triangular (upper trapezoidal if A->nrow < A->ncol).
```

```
See supermatrix.h for the definition of 'SuperMatrix' structure.

Arguments
=========

refact (input) char*
         Specifies whether we want to use perm_r from a previous factor.
         = 'Y': re-use perm_r; perm_r is input, unchanged on exit.
         = 'N': perm_r is determined by partial pivoting, and output.

A        (input) SuperMatrix*
         Original matrix A, permuted by columns, of dimension
         (A->nrow, A->ncol). The type of A can be:
         Stype = NCP; Dtype = D; Mtype = GE.

diag_pivot_thresh (input) float
         Diagonal pivoting threshold. At step j of the Gaussian elimination,
         if abs(A_jj) >= thresh * (max_(i>=j) abs(A_ij)), use A_jj as pivot.
         0 <= thresh <= 1. The default value of thresh is 1, corresponding
         to partial pivoting.

drop_tol (input) float (NOT IMPLEMENTED)
         Drop tolerance parameter. At step j of the Gaussian elimination,
         if abs(A_ij)/(max_i abs(A_ij)) < drop_tol, drop entry A_ij.
         0 <= drop_tol <= 1. The default value of drop_tol is 0.

relax    (input) int
         To control degree of relaxing supernodes. If the number
         of nodes (columns) in a subtree of the elimination tree is less
         than relax, this subtree is considered as one supernode,
         regardless of the row structures of those columns.

panel_size (input) int
         A panel consists of at most panel_size consecutive columns.

etree    (input) int*, dimension (A->ncol)
         Elimination tree of A'*A
         Note: etree is a vector of parent pointers for a forest whose
         vertices are the integers 0 to A->ncol-1; etree[root]==A->ncol.
         On input, the columns of A should be permuted so that the
         etree is in a certain postorder.

work     (input/output) void*, size (lwork) (in bytes)
         User-supplied work space and space for the output data structures.
         Not referenced if lwork = 0;
```

```
lwork    (input) int
         Specifies the size of work array in bytes.
         = 0:  allocate space internally by system malloc;
         > 0:  use user-supplied work array of length lwork in bytes,
               returns error if space runs out.
         = -1: the routine guesses the amount of space needed without
               performing the factorization, and returns it in
               *info; no other side effects.

perm_r   (input/output) int*, dimension (A->nrow)
         Row permutation vector which defines the permutation matrix Pr;
         perm_r[i] = j means row i of A is in position j in Pr*A.
         If refact is not 'Y', perm_r is output argument;
         If refact = 'Y', the pivoting routine will try to use the input
         perm_r, unless a certain threshold criterion is violated.
         In that case, perm_r is overwritten by a new permutation
         determined by partial pivoting or diagonal threshold pivoting.

perm_c   (input) int*, dimension (A->ncol)
         Column permutation vector, which defines the
         permutation matrix Pc; perm_c[i] = j means column i of A is
         in position j in A*Pc.
         When searching for diagonal, perm_c[*] is applied to the
         row subscripts of A, so that diagonal threshold pivoting
         can find the diagonal of A, rather than that of A*Pc.

L        (output) SuperMatrix*
         The factor L from the factorization Pr*A=L*U; use compressed row
         subscripts storage for supernodes, i.e., L has type:
         Stype = SC, Dtype = _S, Mtype = TL.

U        (output) SuperMatrix*
         The factor U from the factorization Pr*A*Pc=L*U. Use column-wise
         storage scheme, i.e., U has types: Stype = NC, Dtype = _S,
         Mtype = TU.

info     (output) int*
         = 0: successful exit
         < 0: if info = -i, the i-th argument had an illegal value
         > 0: if info = i, and i is
            <= A->ncol: U(i,i) is exactly zero. The factorization has
               been completed, but the factor U is exactly singular,
               and division by zero will occur if it is used to solve a
               system of equations.
            > A->ncol: number of bytes allocated when memory allocation
               failure occurred, plus A->ncol. If lwork = -1, it is
               the estimated amount of space needed, plus A->ncol.
```

A.7 SGSTRS

```
void
sgstrs(char *trans, SuperMatrix *L, SuperMatrix *U,
       int *perm_r, int *perm_c, SuperMatrix *B, int *info)

    Purpose
    =======

    SGSTRS solves a system of linear equations A*X=B or A'*X=B
    with A sparse and B dense, using the LU factorization computed by
    sgstrf().

    See supermatrix.h for the definition of 'SuperMatrix' structure.

    Arguments
    =========

    trans   (input) Specifies the form of the system of equations:
             = 'N':  A * X = B  (No transpose)
             = 'T':  A'* X = B  (Transpose)

    L       (input) SuperMatrix*
            The factor L from the factorization Pr*A*Pc=L*U as computed by
            sgstrf(). Use compressed row subscripts storage for supernodes,
            i.e., L has types: Stype = SC, Dtype = _S, Mtype = TL.

    U       (input) SuperMatrix*
            The factor U from the factorization Pr*A*Pc=L*U as computed by
            sgstrf(). Use column-wise storage scheme, i.e., U has types:
            Stype = NC, Dtype = _S, Mtype = TU.

    perm_r  (input) int*, dimension (L->nrow)
            Row permutation vector, which defines the permutation matrix Pr;
            perm_r[i] = j means row i of A is in position j in PrA.

    perm_c  (input) int*, dimension (L->ncol)
            Column permutation vector, which defines the
            permutation matrix Pc; perm_c[i] = j means column i of A is
            in position j in A*Pc.

    B       (input/output) SuperMatrix*
            B has types: Stype = DN, Dtype = _S, Mtype = GE.
            On entry, the right hand side matrix.
            On exit, the solution matrix if info = 0;
```

```
    info    (output) int*
            = 0: successful exit
            < 0: if info = -i, the i-th argument had an illegal value
```

A.8 SLAQGS

```
void
slaqgs(SuperMatrix *A, float *r, float *c, float rowcnd, float colcnd,
       float amax, char *equed)

    Purpose
    =======

    SLAQGS equilibrates a general sparse M by N matrix A using the row and
    scaling factors in the vectors R and C.

    See supermatrix.h for the definition of 'SuperMatrix' structure.

    Arguments
    =========

    A       (input/output) SuperMatrix*
            On exit, the equilibrated matrix.  See EQUED for the form of
            the equilibrated matrix. The type of A can be:
            Stype = NC; Dtype = _S; Mtype = GE.

    R       (input) float*, dimension (A->nrow)
            The row scale factors for A.

    C       (input) float*, dimension (A->ncol)
            The column scale factors for A.

    rowcnd  (input) float
            Ratio of the smallest R(i) to the largest R(i).

    colcnd  (input) float
            Ratio of the smallest C(i) to the largest C(i).

    amax    (input) float
            Absolute value of largest matrix entry.

    equed   (output) char*
            Specifies the form of equilibration that was done.
            = 'N':  No equilibration
            = 'R':  Row equilibration, i.e., A has been premultiplied by
```

```
        diag(R).
= 'C':  Column equilibration, i.e., A has been postmultiplied
        by diag(C).
= 'B':  Both row and column equilibration, i.e., A has been
        replaced by diag(R) * A * diag(C).
```