Sparse Reference Manual

1.4b

Generated by Doxygen 1.2.17

Mon Jun 30 12:01:27 2003

Contents

Chapter 1

Sparse Compound Index

1.1 Sparse Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

Sparse File Index

2.1 Sparse File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Sparse Class Documentation

3.1 spTemplate Struct Reference

`#include <spMatrix.h>`

Public Attributes

- **spElement * Element1**
- **spElement * Element2**
- **spElement * Element3Negated**
- **spElement * Element4Negated**

3.1.1 Detailed Description

This data structure is used to hold pointers to four related elements in matrix. It is used in conjunction with the routines **spGetAdmittance**() (p. 11), **spGetQuad**() (p. 13), and **spGetOnes**() (p. 12). These routines stuff the structure which is later used by the *spADD_QUAD* macro functions above. It is also possible for the user to collect four pointers returned by **spGetElement**() (p. 12) and stuff them into the template. The *spADD_QUAD* routines stuff data into the matrix in locations specified by *Element1* and *Element2* without changing the data. The data is negated before being placed in *Element3* and *Element4*.

The documentation for this struct was generated from the following file:

- **spMatrix.h**

Chapter 4

Sparse File Documentation

4.1 spAllocate.c File Reference

```
#include <stdio.h>
#include "spConfig.h"
#include "spMatrix.h"
#include "spDefs.h"
```

Functions

- **spMatrix spCreate** (int Size, int Complex, **spError** *pError)
- ElementPtr **spcGetElement** (MatrixPtr Matrix)
- ElementPtr **spcGetFillin** (MatrixPtr Matrix)
- void **spDestroy** (**spMatrix** eMatrix)
- **spError spErrorState** (**spMatrix** eMatrix)
- void **spWhereSingular** (**spMatrix** eMatrix, int *pRow, int *pCol)
- int **spGetSize** (**spMatrix** eMatrix, int External)
- void **spSetReal** (**spMatrix** eMatrix)
- void **spSetComplex** (**spMatrix** eMatrix)
- int **spFillinCount** (**spMatrix** eMatrix)
- int **spElementCount** (**spMatrix** eMatrix)

Variables

- char **spcMatrixIsNotValid** [] = "Matrix passed to Sparse is not valid"
- char **spcErrorsMustBeCleared** [] = "Error not cleared"
- char **spcMatrixMustBeFactored** [] = "Matrix must be factored"
- char **spcMatrixMustNotBeFactored** [] = "Matrix must not be factored"

4.1.1 Detailed Description

This file contains functions for allocating and freeing matrices, configuring them, and for accessing global information about the matrix (size, error status, etc.).

Objects that begin with the *spc* prefix are considered private and should not be used.

Author:
Kenneth S. Kundert <kundert@users.sourceforge.net>

4.1.2 Function Documentation

4.1.2.1 spMatrix spCreate (int *Size*, int *Complex*, spError * *pError*)

Allocates and initializes the data structures associated with a matrix.

Returns :
A pointer to the matrix is returned cast into **spMatrix** (typically a pointer to a void). This pointer is then passed and used by the other matrix routines to refer to a particular matrix. If an error occurs, the *NULL* pointer is returned.

Parameters:
Size Size of matrix or estimate of size of matrix if matrix is *EXPANDABLE*.

Complex Type of matrix. If *Complex* is 0 then the matrix is real, otherwise the matrix will be complex. Note that if the routines are not set up to handle the type of matrix requested, then an *spPANIC* error will occur. Further note that if a matrix will be both real and complex, it must be specified here as being complex.

pError Returns error flag, needed because function **spErrorState***()* (p. 8) will not work correctly if **spCreate***()* (p. 8) returns *NULL*. Possible errors include *spNO_MEMORY* and *spPANIC*.

4.1.2.2 void spDestroy (spMatrix *eMatrix*)

Destroys a matrix and frees all memory associated with it.

Parameters:
eMatrix Pointer to the matrix frame which is to be destroyed.

4.1.2.3 int spElementCount (spMatrix *eMatrix*)

This function returns the total number of elements (including fill-ins) that currently exists in a matrix.

Parameters:
eMatrix Pointer to matrix.

4.1.2.4 spError spErrorState (spMatrix *eMatrix*)

This function returns the error status of the given matrix.

Returns :
The error status of the given matrix.

Parameters:
eMatrix The pointer to the matrix for which the error status is desired.

4.1.2.5 int spFillinCount (spMatrix *eMatrix*)

This function returns the number of fill-ins that currently exists in a matrix.

Parameters:

eMatrix Pointer to matrix.

4.1.2.6 int spGetSize (spMatrix *eMatrix*, int *External*)

Returns the size of the matrix. Either the internal or external size of the matrix is returned.

Parameters:

eMatrix Pointer to matrix.

External If *External* is set true, the external size , i.e., the value of the largest external row or column number encountered is returned. Otherwise the true size of the matrix is returned. These two sizes may differ if the *TRANSLATE* option is set true.

4.1.2.7 void spSetComplex (spMatrix *eMatrix*)

Forces matrix to be complex.

Parameters:

eMatrix Pointer to matrix.

4.1.2.8 void spSetReal (spMatrix *eMatrix*)

Forces matrix to be real.

Parameters:

eMatrix Pointer to matrix.

4.1.2.9 void spWhereSingular (spMatrix *eMatrix*, int * *pRow*, int * *pCol*)

This function returns the row and column number where the matrix was detected as singular (if pivoting was allowed on the last factorization) or where a zero was detected on the diagonal (if pivoting was not allowed on the last factorization). Pivoting is performed only in **spOrderAndFactor**() (p. 24).

Parameters:

eMatrix The matrix for which the error status is desired.

pRow The row number.

pCol The column number.

4.2 spBuild.c File Reference

```
#include <stdio.h>
#include "spConfig.h"
#include "spMatrix.h"
#include "spDefs.h"
```

Defines

- #define **spINSIDE_SPARSE**
- #define **BorderRight** 0
- #define **BorderDown** 1
- #define **DiagRight** 2
- #define **DiagDown** 3

Functions

- void **spClear** (**spMatrix** eMatrix)
- **spElement** * **spFindElement** (**spMatrix** eMatrix, int Row, int Col)
- **spElement** * **spGetElement** (**spMatrix** eMatrix, int Row, int Col)
- **spError spGetAdmittance** (**spMatrix** Matrix, int Node1, int Node2, struct **spTemplate** *Template)
- **spError spGetQuad** (**spMatrix** Matrix, int Row1, int Row2, int Col1, int Col2, struct **spTemplate** *Template)
- **spError spGetOnes** (**spMatrix** Matrix, int Pos, int Neg, int Eqn, struct **spTemplate** *Template)
- ElementPtr **spcFindDiag** (MatrixPtr Matrix, register int Index)
- ElementPtr **spcCreateElement** (MatrixPtr Matrix, int Row, register int Col, register ElementPtr *ppToLeft, register ElementPtr *ppAbove, BOOLEAN Fillin)
- void **spcLinkRows** (MatrixPtr Matrix)
- int **spInitialize** (**spMatrix** eMatrix, int(*pInit)(**spElement** *pElement, spGenericPtr pInitInfo, int Row, int Col))
- void **spInstallInitInfo** (**spElement** *pElement, spGenericPtr pInitInfo)
- spGenericPtr **spGetInitInfo** (**spElement** *pElement)

4.2.1 Detailed Description

This file contains the routines associated with clearing, loading and preprocessing the matrix.

Objects that begin with the *spc* prefix are considered private and should not be used.

Author:
Kenneth S. Kundert <kundert@users.sourceforge.net>

4.2.2 Function Documentation

4.2.2.1 void spClear (spMatrix *eMatrix*)

Sets every element of the matrix to zero and clears the error flag.

Parameters:

eMatrix Pointer to matrix that is to be cleared.

4.2.2.2 spElement* spFindElement (spMatrix *eMatrix*, int *Row*, int *Col*)

This routine is used to find an element given its indices. It will not create it if it does not exist.

Returns :

A pointer to the desired element, or *NULL* if it does not exist.

Parameters:

eMatrix Pointer to matrix.

Row Row index for element.

Col Column index for element.

See also:

spGetElement() (p. 12)

4.2.2.3 spError spGetAdmittance (spMatrix *Matrix*, int *Node1*, int *Node2*, struct spTemplate * *Template*)

Performs same function as **spGetElement**() (p. 12) except rather than one element, all four matrix elements for a floating two terminal admittance component are added. This routine also works if component is grounded. Positive elements are placed at [Node1,Node2] and [Node2,Node1]. This routine is only to be used after **spCreate**() (p. 8) and before **spMNA_Preorder**() (p. 72), **spFactor**() (p. 23) or **spOrderAndFactor**() (p. 24).

Returns :

Error code. Possible errors include *spNO_MEMORY*. Error is not cleared in this routine.

Parameters:

Matrix Pointer to the matrix that component is to be entered in.

Node1 Row and column indices for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Node zero is the ground node. In no case may *Node1* be less than zero.

Node2 Row and column indices for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Node zero is the ground node. In no case may *Node2* be less than zero.

Template Collection of pointers to four elements that are later used to directly address elements. User must supply the template, this routine will fill it.

4.2.2.4 spElement* spGetElement (spMatrix *eMatrix*, int *Row*, int *Col*)

Finds element [Row,Col] and returns a pointer to it. If element is not found then it is created and spliced into matrix. This routine is only to be used after **spCreate**() (p. 8) and before **spMNA_Preorder**() (p. 72), **spFactor**() (p. 23) or **spOrderAndFactor**() (p. 24). Returns a pointer to the real portion of an **spElement**. This pointer is later used by *spADD_xxx_ELEMENT* to directly access element.

Returns :

Returns a pointer to the element. This pointer is then used to directly access the element during successive builds.

Parameters:

eMatrix Pointer to the matrix that the element is to be added to.

Row Row index for element. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Elements placed in row zero are discarded. In no case may *Row* be less than zero.

Col Column index for element. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Elements placed in column zero are discarded. In no case may *Col* be less than zero.

See also:

spFindElement() (p. 11)

4.2.2.5 spGenericPtr spGetInitInfo (spElement * *pElement*)

This function returns a pointer to a data structure that is used to contain initialization information to a matrix element.

Returns :

The pointer to the initialiation information data structure that is associated with a particular matrix element.

Parameters:

pElement Pointer to the matrix element.

See also:

spInitialize() (p. 14)

4.2.2.6 spError spGetOnes (spMatrix *Matrix*, int *Pos*, int *Neg*, int *Eqn*, struct spTemplate * *Template*)

Addition of four structural ones to matrix by index. Performs similar function to **spGetQuad**() (p. 13) except this routine is meant for components that do not have an admittance representation.

The following stamp is used:

```
       Pos  Neg  Eqn
Pos  [  .    .    1  ]
Neg  [  .    .   -1  ]
Eqn  [  1   -1    .  ]
```

Returns :
Error code. Possible errors include *spNO_MEMORY*. Error is not cleared in this routine.

Parameters:

Matrix Pointer to the matrix that component is to be entered in.

Pos See stamp above. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may *Pos* be less than zero.

Neg See stamp above. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may *Neg* be less than zero.

Eqn See stamp above. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may *Eqn* be less than zero.

Template Collection of pointers to four elements that are later used to directly address elements. User must supply the template, this routine will fill it.

4.2.2.7 spError spGetQuad (spMatrix *Matrix*, int *Row1*, int *Row2*, int *Col1*, int *Col2*, struct spTemplate * *Template*)

Similar to **spGetAdmittance**() (p. 11), except that **spGetAdmittance**() (p. 11) only handles 2-terminal components, whereas **spGetQuad**() (p. 13) handles simple 4-terminals as well. These 4-terminals are simply generalized 2-terminals with the option of having the sense terminals different from the source and sink terminals. **spGetQuad**() (p. 13) adds four elements to the matrix. Positive elements occur at [Row1,Col1] [Row2,Col2] while negative elements occur at [Row1,Col2] and [Row2,Col1]. The routine works fine if any of the rows and columns are zero. This routine is only to be used after **spCreate**() (p. 8) and before **spMNA_Preorder**() (p. 72), **spFactor**() (p. 23) or **spOrderAndFactor**() (p. 24) unless *TRANSLATE* is set true.

Returns :
Error code. Possible errors include *spNO_MEMORY*. Error is not cleared in this routine.

Parameters:

Matrix Pointer to the matrix that component is to be entered in.

Row1 First row index for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may Row1 be less than zero.

Row2 Second row index for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may Row2 be less than zero.

Col1 First column index for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground column. In no case may Col1 be less than zero.

Col2 Second column index for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground column. In no case may Col2 be less than zero.

Template Collection of pointers to four elements that are later used to directly address elements. User must supply the template, this routine will fill it.

4.2.2.8 int spInitialize (spMatrix *eMatrix*, int(* *pInit*)(spElement *pElement, spGenericPtr pInitInfo, int Row, int Col))

Initialize the matrix.

With the *INITIALIZE* compiler option (see **spConfig.h**) set true, Sparse allows the user to keep initialization information with each structurally nonzero matrix element. Each element has a pointer that is set and used by the user. The user can set this pointer using **spInstallInitInfo**() (p. 14) and may be read using **spGetInitInfo**() (p. 12). Both may be used only after the element exists. The function **spInitialize**() (p. 14) is a user customizable way to initialize the matrix. Passed to this routine is a function pointer. **spInitialize**() (p. 14) sweeps through every element in the matrix and checks the *pInitInfo* pointer (the user supplied pointer). If the *pInitInfo* is *NULL*, which is true unless the user changes it (almost always true for fill-ins), then the element is zeroed. Otherwise, the function pointer is called and passed the *pInitInfo* pointer as well as the element pointer and the external row and column numbers. If the user sets the value of each element, then **spInitialize**() (p. 14) replaces **spClear**() (p. 11).

The user function is expected to return a nonzero integer if there is a fatal error and zero otherwise. Upon encountering a nonzero return code, **spInitialize**() (p. 14) terminates, sets the error state of the matrix to be *spMANGLED*, and returns the error code.

Returns :
Returns the return value of the *pInit()* function.

Parameters:
eMatrix Pointer to matrix.

pInit Pointer to a function that initializes an element.

See also:
spClear() (p. 11)

4.2.2.9 void spInstallInitInfo (spElement * *pElement*, spGenericPtr *pInitInfo*)

This function installs a pointer to a data structure that is used to contain initialization information to a matrix element. It is is then used by **spInitialize**() (p. 14) to initialize the matrix.

Parameters:
pElement Pointer to matrix element.

pInitInfo Pointer to the data structure that will contain initialiation information.

See also:
spInitialize() (p. 14)

4.3 spConfig.h File Reference

```
#include <limits.h>
#include <float.h>
```

Defines

- #define **REAL** YES
- #define **EXPANDABLE** YES
- #define **TRANSLATE** YES
- #define **INITIALIZE** YES
- #define **DIAGONAL_PIVOTING** YES
- #define **ARRAY_OFFSET** NOT FORTRAN
- #define **MODIFIED_MARKOWITZ** NO
- #define **DELETE** YES
- #define **STRIP** YES
- #define **MODIFIED_NODAL** YES
- #define **QUAD_ELEMENT** YES
- #define **TRANSPOSE** YES
- #define **SCALING** YES
- #define **DOCUMENTATION** YES
- #define **MULTIPLICATION** YES
- #define **DETERMINANT** YES
- #define **STABILITY** YES
- #define **CONDITION** YES
- #define **PSEUDOCONDITION** YES
- #define **FORTRAN** YES
- #define **DEBUG** YES
- #define **spCOMPLEX** 1
- #define **spSEPARATED_COMPLEX_VECTORS** 0
- #define **DEFAULT_THRESHOLD** 1.0e-3
- #define **DIAG_PIVOTING_AS_DEFAULT** YES
- #define **SPACE_FOR_ELEMENTS** 6
- #define **SPACE_FOR_FILL_INS** 4
- #define **ELEMENTS_PER_ALLOCATION** 31
- #define **MINIMUM_ALLOCATED_SIZE** 6
- #define **EXPANSION_FACTOR** 1.5
- #define **MAX_MARKOWITZ_TIES** 100
- #define **TIES_MULTIPLIER** 5
- #define **DEFAULT_PARTITION** spAUTO_PARTITION
- #define **PRINTER_WIDTH** 80
- #define **spcCONCAT**(prefix, suffix) prefix/**/suffix
- #define **spcQUOTE**(x) "x"
- #define **spcFUNC_NEEDS_FILE**(func, file) func/**/_requires_/**/file/**/_to_be_included_
- #define **spcEXTERN** extern
- #define **spcNO_ARGS**
- #define **spcCONST**
- #define **MACHINE_RESOLUTION** DBL_EPSILON

- #define **LARGEST_REAL** DBL_MAX
- #define **SMALLEST_REAL** DBL_MIN
- #define **LARGEST_SHORT_INTEGER** SHRT_MAX
- #define **LARGEST_LONG_INTEGER** LONG_MAX
- #define **ANNOTATE** NONE
- #define **NONE** 0
- #define **ON_STRANGE_BEHAVIOR** 1
- #define **FULL** 2

Typedefs

- typedef char * **spGenericPtr**

4.3.1 Detailed Description

This file contains macros for the sparse matrix routines that are used to define the personality of the routines. The user is expected to modify this file to maximize the performance of the routines with his/her matrices.

Macros are distinguished by using solely capital letters in their identifiers. This contrasts with C defined identifiers which are strictly lower case, and program variable and procedure names which use both upper and lower case.

Objects that begin with the *spc* prefix are considered private and should not be used.

Author:
Kenneth S. Kundert <kundert@users.sourceforge.net>

4.3.2 Define Documentation

4.3.2.1 #define ANNOTATE NONE

This macro changes the amount of annotation produced by the matrix routines. The annotation is used as a debugging aid. Change the number associated with *ANNOTATE* to change the amount of annotation produced by the program. Possible values include *NONE*, *ON_STRANGE_BEHAVIOR*, and *FULL*. *NONE* is recommended.

4.3.2.2 #define ARRAY_OFFSET NOT FORTRAN

This determines whether arrays start at an index of zero or one. This option is necessitated by the fact that standard C convention dictates that arrays begin with an index of zero but the standard mathematic convention states that arrays begin with an index of one. So if you prefer to start your arrays with zero, or your calling Sparse from FORTRAN, set ARRAY_OFFSET to NO or 0. Otherwise, set ARRAY_OFFSET to YES or 1. Note that if you use an offset of one, the arrays that you pass to Sparse must have an allocated length of one plus the size of the matrix. ARRAY_OFFSET must be either 0 or 1, no other offsets are valid.

4.3.2.3 #define CONDITION YES

This specifies that **spCondition**() (p. 69) and **spNorm**() (p. 74), the code that computes a good estimate of the condition number of the matrix, should be compiled. Recomend *NO*.

4.3.2.4 #define DEBUG YES

This specifies that additional error checking will be compiled. The type of error checked are those that are common when the matrix routines are first integrated into a user's program. Once the routines have been integrated in and are running smoothly, this option should be turned off. *YES* is recommended.

4.3.2.5 #define DEFAULT_PARTITION spAUTO_PARTITION

Which partition mode is used by **spPartition**() (p. 25) as default. Possibilities include *spDIRECT_PARTITION* (each row used direct addressing, best for a few relatively dense matrices), *spINDIRECT_PARTITION* (each row used indirect addressing, best for a few very sparse matrices), and *spAUTO_PARTITION* (direct or indirect addressing is chosen on a row-by-row basis, carries a large overhead, but speeds up both dense and sparse matrices, best if there is a large number of matrices that can use the same ordering.

4.3.2.6 #define DEFAULT_THRESHOLD 1.0e-3

The relative threshold used if the user enters an invalid threshold. Also the threshold used by **spFactor**() (p. 23) when calling **spOrderAndFactor**() (p. 24). The default threshold should not be less than or equal to zero nor larger than one. 0.001 is recommended.

4.3.2.7 #define DELETE YES

This specifies that the **spDeleteRowAndCol**() (p. 70) routine should be compiled. Note that for this routine to be compiled, both *DELETE* and *TRANSLATE* should be set true.

4.3.2.8 #define DETERMINANT YES

This specifies that the routine **spDeterminant**() (p. 70) should be complied.

4.3.2.9 #define DIAG_PIVOTING_AS_DEFAULT YES

This indicates whether **spOrderAndFactor**() (p. 24) should use diagonal pivoting as default. This issue only arises when **spOrderAndFactor**() (p. 24) is called from **spFactor**() (p. 23). *YES* is recommended.

4.3.2.10 #define DIAGONAL_PIVOTING YES

Many matrices, and in particular node- and modified-node admittance matrices, tend to be nearly symmetric and nearly diagonally dominant. For these matrices, it is a good idea to select pivots from the diagonal. With this option enabled, this is exactly what happens, though if no satisfactory pivot can be found on the diagonal, an off-diagonal pivot will be used. If this option is disabled, Sparse does not preferentially search the diagonal. Because of this, Sparse has a wider variety of pivot candidates available, and so presumably fewer fill-ins will be created. However, the initial pivot selection process will take considerably longer. If working with node admittance matrices, or other matrices with a strong diagonal, it is probably best to use *DIAGONAL_PIVOTING* for two reasons. First, accuracy will be better because pivots will be chosen from the large diagonal elements, thus reducing the chance of growth. Second, a near optimal ordering will be chosen quickly. If the class of matrices you are working with does not have a strong diagonal, do not use *DIAGONAL_PIVOTING*, but consider using

a larger threshold. When *DIAGONAL_PIVOTING* is turned off, the following options and constants are not used: *MODIFIED_MARKOWITZ*, *MAX_MARKOWITZ_TIES*, and *TIES_MULTIPLIER*.

4.3.2.11 #define DOCUMENTATION YES

This specifies that routines that are used to document the matrix, such as **spPrint**() (p. 66) and **spFileMatrix**() (p. 65), should be compiled.

4.3.2.12 #define ELEMENTS_PER_ALLOCATION 31

The number of matrix elements requested from the malloc utility on each call to it. Setting this value greater than 1 reduces the amount of overhead spent in this system call. On a virtual memory machine, its good to allocate slightly less than a page worth of elements at a time (or some multiple thereof). 31 is recommended.

4.3.2.13 #define EXPANDABLE YES

Setting this compiler flag true (1) makes the matrix expandable before it has been factored. If the matrix is expandable, then if an element is added that would be considered out of bounds in the current matrix, the size of the matrix is increased to hold that element. As a result, the size of the matrix need not be known before the matrix is built. The matrix can be allocated with size zero and expanded.

4.3.2.14 #define EXPANSION_FACTOR 1.5

The amount the allocated size of the matrix is increased when it is expanded.

4.3.2.15 #define FORTRAN YES

This specifies that the *FORTRAN* interface routines should be compiled. When interfacing to *FORTRAN* programs, the *ARRAY_OFFSET* options should be set to NO.

4.3.2.16 #define FULL 2

A possible value for *ANNOTATE*. Enables full annotation.

4.3.2.17 #define INITIALIZE YES

Causes the **spInitialize**() (p. 14), **spGetInitInfo**() (p. 12), and **spInstallInitInfo**() (p. 14) routines to be compiled. These routines allow the user to store and read one pointer in each nonzero element in the matrix. **spInitialize**() (p. 14) then calls a user specified function for each structural nonzero in the matrix, and includes this pointer as well as the external row and column numbers as arguments. This allows the user to write custom matrix initialization routines.

4.3.2.18 #define LARGEST_LONG_INTEGER LONG_MAX

The largest possible value of longs.

4.3.2.19 #define LARGEST_REAL DBL_MAX

The largest possible value of spREAL.

4.3.2.20 #define LARGEST_SHORT_INTEGER SHRT_MAX

The largest possible value of shorts.

4.3.2.21 #define MACHINE_RESOLUTION DBL_EPSILON

The resolution of spREAL.

4.3.2.22 #define MAX_MARKOWITZ_TIES 100

Some terminology should be defined. The Markowitz row count is the number of non-zero elements in a row excluding the one being considered as pivot. There is one Markowitz row count for every row. The Markowitz column is defined similarly for columns. The Markowitz product for an element is the product of its row and column counts. It is a measure of how much work would be required on the next step of the factorization if that element were chosen to be pivot. A small Markowitz product is desirable.

This number is used for two slightly different things, both of which relate to the search for the best pivot. First, it is the maximum number of elements that are Markowitz tied that will be sifted through when trying to find the one that is numerically the best. Second, it creates an upper bound on how large a Markowitz product can be before it eliminates the possibility of early termination of the pivot search. In other words, if the product of the smallest Markowitz product yet found and *TIES_MULTIPLIER* is greater than *MAX_MARKOWITZ_TIES*, then no early termination takes place. Set *MAX_MARKOWITZ_TIES* to some small value if no early termination of the pivot search is desired. An array of RealNumbers is allocated of size *MAX_MARKOWITZ_TIES* so it must be positive and shouldn't be too large. Active when MODIFIED_MARKOWITZ is 1 (YES). 100 is recommended.

See also:
TIES_MULTIPLIER (p. 22)

4.3.2.23 #define MINIMUM_ALLOCATED_SIZE 6

The minimum allocated size of a matrix. Note that this does not limit the minimum size of a matrix. This just prevents having to resize a matrix many times if the matrix is expandable, large and allocated with an estimated size of zero. This number should not be less than one.

4.3.2.24 #define MODIFIED_MARKOWITZ NO

This specifies that the modified Markowitz method of pivot selection is to be used. The modified Markowitz method differs from standard Markowitz in two ways. First, under modified Markowitz, the search for a pivot can be terminated early if a adequate (in terms of sparsity) pivot candidate is found. Thus, when using modified Markowitz, the initial factorization can be faster, but at the expense of a suboptimal pivoting order that may slow subsequent factorizations. The second difference is in the way modified Markowitz breaks Markowitz ties. When two or more elements are pivot candidates and they all have the same Markowitz product, then the tie is broken by choosing the element that is best numerically. The numerically best element is the one with the largest ratio of its magnitude to the

magnitude of the largest element in the same column, excluding itself. The modified Markowitz method results in marginally better accuracy. This option is most appropriate for use when working with very large matrices where the initial factor time represents an unacceptable burden. *NO* is recommended.

4.3.2.25 #define MODIFIED_NODAL YES

This specifies that the routine that preorders modified node admittance matrices should be compiled. This routine results in greater speed and accuracy if used with this type of matrix.

4.3.2.26 #define MULTIPLICATION YES

This specifies that routines that are used to multily the matrix by a vector, such as **spMultiply**() (p. 73) and **spMultTransposed**() (p. 74), should be compiled.

4.3.2.27 #define NONE 0

A possible value for *ANNOTATE*. Disables all annotation.

4.3.2.28 #define ON_STRANGE_BEHAVIOR 1

A possible value for *ANNOTATE*. Causes annotation to be produce upon unusual occurances only.

4.3.2.29 #define PRINTER_WIDTH 80

The number of characters per page width. Set to 80 for terminal, 132 for line printer. Controls how many columns printed by **spPrint**() (p. 66) per page width.

4.3.2.30 #define PSEUDOCONDITION YES

This specifies that **spPseudoCondition**() (p. 74), the code that computes a crude and easily fooled indicator of ill-conditioning in the matrix, should be compiled. Recomend *NO*.

4.3.2.31 #define QUAD_ELEMENT YES

This specifies that the routines that allow four related elements to be entered into the matrix at once should be compiled. These elements are usually related to an admittance. The routines affected by *QUAD_ELEMENT* are the **spGetAdmittance**() (p. 11), **spGetQuad**() (p. 13) and **spGetOnes**() (p. 12) routines.

4.3.2.32 #define REAL YES

This specifies that the routines are expected to handle real systems of equations. The routines can be compiled to handle both real and complex systems at the same time, but there is a slight speed and memory advantage if the routines are complied to handle only real systems of equations.

4.3.2.33 #define SCALING YES

This specifies that the routine that performs scaling on the matrix should be complied. Scaling is not strongly supported. The routine to scale the matrix is provided, but no routines are provided to scale and descale the RHS and Solution vectors. It is suggested that if scaling is desired, it only be preformed when the pivot order is being chosen [in **spOrderAndFactor**() (p. 24)]. This is the only time scaling has an effect. The scaling may then either be removed from the solution by the user or the scaled factors may simply be thrown away. *NO* is recommended.

4.3.2.34 #define SMALLEST_REAL DBL_MIN

The smalles possible positive value of spREAL.

4.3.2.35 #define SPACE_FOR_ELEMENTS 6

This number multiplied by the size of the matrix equals the number of elements for which memory is initially allocated in **spCreate**() (p. 8). 6 is recommended.

4.3.2.36 #define SPACE_FOR_FILL_INS 4

This number multiplied by the size of the matrix equals the number of elements for which memory is initially allocated and specifically reserved for fill-ins in **spCreate**() (p. 8). 4 is recommended.

4.3.2.37 #define spCOMPLEX 1

This specifies that the routines will be complied to handle complex systems of equations.

4.3.2.38 #define spSEPARATED_COMPLEX_VECTORS 0

This specifies the format for complex vectors. If this is set false then a complex vector is made up of one double sized array of RealNumber's in which the real and imaginary numbers are placed alternately in the array. In other words, the first entry would be Complex[1].Real, then comes Complex[1].Imag, then Complex[2].Real, etc. If *spSEPARATED_COMPLEX_VECTORS* is set true, then each complex vector is represented by two arrays of *spREALs*, one with the real terms, the other with the imaginary. *NO* is recommended.

4.3.2.39 #define STABILITY YES

This specifies that **spLargestElement**() (p. 71) and **spRoundoff**() (p. 75) should be compiled. These routines are used to check the stability (and hence the quality of the pivoting) of the factorization by computing a bound on the size of the element is the matrix $E = A - LU$. If this bound is very high after applying **spOrderAndFactor**() (p. 24), then the pivot threshold should be raised. If the bound increases greatly after using **spFactor**() (p. 23), then the matrix should probably be reordered. Recomend *NO*.

4.3.2.40 #define STRIP YES

This specifies that the **spStripFills**() (p. 75) routine should be compiled.

4.3.2.41 #define TIES_MULTIPLIER 5

Specifies the number of Markowitz ties that are allowed to occur before the search for the pivot is terminated early. Set to some large value if no early termination of the pivot search is desired. This number is multiplied times the Markowitz product to determine how many ties are required for early termination. This means that more elements will be searched before early termination if a large number of fill-ins could be created by accepting what is currently considered the best choice for the pivot. Active when *MODIFIED_MARKOWITZ* is 1 (YES). Setting this number to zero effectively eliminates all pivoting, which should be avoided. This number must be positive. *TIES_MULTIPLIER* is also used when diagonal pivoting breaks down. 5 is recommended.

See also:
MAX_MARKOWITZ_TIES (p. 19)

4.3.2.42 #define TRANSLATE YES

This option allows the set of external row and column numbers to be non-packed. In other words, the row and column numbers do not have to be contiguous. The priced paid for this flexibility is that when *TRANSLATE* is set true, the time required to initially build the matrix will be greater because the external row and column number must be translated into internal equivalents. This translation brings about other benefits though. First, the **spGetElement**() (p. 12) and **spGetAdmittance**() (p. 11) routines may be used after the matrix has been factored. Further, elements, and even rows and columns, may be added to the matrix, and row and columns may be deleted from the matrix, after it has been factored. Note that when the set of row and column number is not a packed set, neither are the *RHS* and *Solution* vectors. Thus the size of these vectors must be at least as large as the external size, which is the value of the largest given row or column numbers.

4.3.2.43 #define TRANSPOSE YES

This specifies that the routines that solve the matrix as if it was transposed should be compiled. These routines are useful when performing sensitivity analysis using the adjoint method.

4.4 spFactor.c File Reference

```
#include <stdio.h>
#include "spConfig.h"
#include "spMatrix.h"
#include "spDefs.h"
```

Functions

- **spError spOrderAndFactor** (**spMatrix** eMatrix, spREAL RHS[], spREAL RelThreshold, spREAL AbsThreshold, int DiagPivoting)
- **spError spFactor** (**spMatrix** eMatrix)
- void **spPartition** (**spMatrix** eMatrix, int Mode)
- void **spcCreateInternalVectors** (MatrixPtr Matrix)
- void **spcRowExchange** (MatrixPtr Matrix, int Row1, int Row2)
- void **spcColExchange** (MatrixPtr Matrix, int Col1, int Col2)

4.4.1 Detailed Description

This file contains the routines to factor the matrix into LU form.

Objects that begin with the *spc* prefix are considered private and should not be used.

Author:
Kenneth S. Kundert <kundert@users.sourceforge.net>

4.4.2 Function Documentation

4.4.2.1 spError spFactor (spMatrix *eMatrix*)

This routine is the companion routine to **spOrderAndFactor**() (p. 24). Unlike **spOrderAndFactor**() (p. 24), **spFactor**() (p. 23) cannot change the ordering. It is also faster than **spOrderAndFactor**() (p. 24). The standard way of using these two routines is to first use **spOrderAndFactor**() (p. 24) for the initial factorization. For subsequent factorizations, **spFactor**() (p. 23) is used if there is some assurance that little growth will occur (say for example, that the matrix is diagonally dominant). If **spFactor**() (p. 23) is called for the initial factorization of the matrix, then **spOrderAndFactor**() (p. 24) is automatically called with the default threshold. This routine uses "row at a time" *LU* factorization. Pivots are associated with the lower triangular matrix and the diagonals of the upper triangular matrix are ones.

Returns :
The error code is returned. Possible errors are *spNO_MEMORY*, *spSINGULAR*, *spZERO_DIAG* and *spSMALL_PIVOT*. Error is cleared upon entering this function.

Parameters:
eMatrix Pointer to matrix.

See also:
spOrderAndFactor() (p. 24)

4.4.2.2 spError spOrderAndFactor (spMatrix *eMatrix*, spREAL *RHS*[ ], spREAL *RelThreshold*, spREAL *AbsThreshold*, int *DiagPivoting*)

This routine chooses a pivot order for the matrix and factors it into *LU* form. It handles both the initial factorization and subsequent factorizations when a reordering is desired. This is handled in a manner that is transparent to the user. The routine uses a variation of Gauss's method where the pivots are associated with *L* and the diagonal terms of *U* are one.

Returns :

The error code is returned. Possible errors are *spNO_MEMORY*, *spSINGULAR* and *spSMALL_PIVOT*. Error is cleared upon entering this function.

Parameters:

eMatrix Pointer to the matrix.

RHS Representative right-hand side vector that is used to determine pivoting order when the right hand side vector is sparse. If RHS is a NULL pointer then the RHS vector is assumed to be full and it is not used when determining the pivoting order.

RelThreshold This number determines what the pivot relative threshold will be. It should be between zero and one. If it is one then the pivoting method becomes complete pivoting, which is very slow and tends to fill up the matrix. If it is set close to zero the pivoting method becomes strict Markowitz with no threshold. The pivot threshold is used to eliminate pivot candidates that would cause excessive element growth if they were used. Element growth is the cause of roundoff error. Element growth occurs even in well-conditioned matrices. Setting the *RelThreshold* large will reduce element growth and roundoff error, but setting it too large will cause execution time to be excessive and will result in a large number of fill-ins. If this occurs, accuracy can actually be degraded because of the large number of operations required on the matrix due to the large number of fill-ins. A good value seems to be 0.001. The default is chosen by giving a value larger than one or less than or equal to zero. This value should be increased and the matrix resolved if growth is found to be excessive. Changing the pivot threshold does not improve performance on matrices where growth is low, as is often the case with ill-conditioned matrices. Once a valid threshold is given, it becomes the new default. The default value of *RelThreshold* was choosen for use with nearly diagonally dominant matrices such as node- and modified-node admittance matrices. For these matrices it is usually best to use diagonal pivoting. For matrices without a strong diagonal, it is usually best to use a larger threshold, such as 0.01 or 0.1.

AbsThreshold The absolute magnitude an element must have to be considered as a pivot candidate, except as a last resort. This number should be set significantly smaller than the smallest diagonal element that is is expected to be placed in the matrix. If there is no reasonable prediction for the lower bound on these elements, then *AbsThreshold* should be set to zero. *AbsThreshold* is used to reduce the possibility of choosing as a pivot an element that has suffered heavy cancellation and as a result mainly consists of roundoff error. Once a valid threshold is given, it becomes the new default.

DiagPivoting A flag indicating that pivot selection should be confined to the diagonal if possible. If *DiagPivoting* is nonzero and if *DIAGONAL_PIVOTING* is enabled pivots will be chosen only from the diagonal unless there are no diagonal elements that satisfy the threshold criteria. Otherwise, the entire reduced submatrix is searched when looking for a pivot. The diagonal pivoting in Sparse is efficient and well refined, while the off-diagonal pivoting is not. For symmetric and near symmetric matrices, it is best to use diagonal pivoting because it results in the best performance when reordering the matrix and when factoring the matrix without ordering. If there is a considerable amount of nonsymmetry in the matrix, then off-diagonal pivoting may result in a better equation ordering simply because there are more pivot candidates to choose from. A better ordering results in faster subsequent factorizations. However, the initial pivot selection process takes considerably longer for off-diagonal pivoting.

See also:
spFactor() (p. 23)

4.4.2.3 void spPartition (spMatrix *eMatrix*, int *Mode*)

This routine determines the cost to factor each row using both direct and indirect addressing and decides, on a row-by-row basis, which addressing mode is fastest. This information is used in **spFactor**() (p. 23) to speed the factorization.

When factoring a previously ordered matrix using **spFactor**() (p. 23), Sparse operates on a row-at-a-time basis. For speed, on each step, the row being updated is copied into a full vector and the operations are performed on that vector. This can be done one of two ways, either using direct addressing or indirect addressing. Direct addressing is fastest when the matrix is relatively dense and indirect addressing is best when the matrix is quite sparse. The user selects the type of partition used with *Mode*. If *Mode* is set to *spDIRECT_PARTITION*, then the all rows are placed in the direct addressing partition. Similarly, if *Mode* is set to *spINDIRECT_PARTITION*, then the all rows are placed in the indirect addressing partition. By setting *Mode* to *spAUTO_PARTITION*, the user allows Sparse to select the partition for each row individually. **spFactor**() (p. 23) generally runs faster if Sparse is allowed to choose its own partitioning, however choosing a partition is expensive. The time required to choose a partition is of the same order of the cost to factor the matrix. If you plan to factor a large number of matrices with the same structure, it is best to let Sparse choose the partition. Otherwise, you should choose the partition based on the predicted density of the matrix.

Parameters:
eMatrix Pointer to matrix.

Mode Mode must be one of three special codes: *spDIRECT_PARTITION*, *spINDIRECT_PARTITION*, or *spAUTO_PARTITION*.

4.5 spFortran.c File Reference

```
#include <stdio.h>
#include "spConfig.h"
#include "spMatrix.h"
#include "spDefs.h"
```

Defines

- #define **spINSIDE_SPARSE**
- #define **sfCreate** sfcreate
- #define **sfStripFills** sfstripfills
- #define **sfDestroy** sfdestroy
- #define **sfClear** sfclear
- #define **sfGetElement** sfgetelement
- #define **sfGetAdmittance** sfgetadmittance
- #define **sfGetQuad** sfgetquad
- #define **sfGetOnes** sfgetones
- #define **sfAdd1Real** sfadd1real
- #define **sfAdd1Imag** sfadd1imag
- #define **sfAdd1Complex** sfadd1complex
- #define **sfAdd4Real** sfadd4real
- #define **sfAdd4Imag** sfadd4imag
- #define **sfAdd4Complex** sfadd4complex
- #define **sfOrderAndFactor** sforderandfactor
- #define **sfFactor** sffactor
- #define **sfPartition** sfpartition
- #define **sfSolve** sfsolve
- #define **sfSolveTransposed** sfsolvetransposed
- #define **sfPrint** sfprint
- #define **sfFileMatrix** sffilematrix
- #define **sfFileVector** sffilevector
- #define **sfFileStats** sffilestats
- #define **sfMNA_Preorder** sfmna_preorder
- #define **sfScale** sfscale
- #define **sfMultiply** sfmultiply
- #define **sfMultTransposed** sfmulttransposed
- #define **sfDeterminant** sfdeterminant
- #define **sfError** sferror
- #define **sfErrorMessage** sferrormessage
- #define **sfWhereSingular** sfwheresingular
- #define **sfGetSize** sfgetsize
- #define **sfSetReal** sfsetreal
- #define **sfSetComplex** sfsetcomplex
- #define **sfFillinCount** sffillincount
- #define **sfElementCount** sfelementcount
- #define **sfDeleteRowAndCol** sfdeleterowandcol
- #define **sfPseudoCondition** sfpseudocondition

- #define **sfCondition** sfcondition
- #define **sfNorm** sfnorm
- #define **sfLargestElement** sflargestelement
- #define **sfRoundoff** sfroundoff
- #define **MATRIX_FILE_NAME** "**spMatrix**"
- #define **STATS_FILE_NAME** "spStats"

Functions

- long **sfCreate** (int *Size, int *Complex, int *Error)
- void **sfDestroy** (long *Matrix)
- void **sfStripFills** (long *Matrix)
- void **sfClear** (long *Matrix)
- long **sfGetElement** (long *Matrix, int *Row, int *Col)
- int **sfGetAdmittance** (long *Matrix, int *Node1, int *Node2, long Template[4])
- int **sfGetQuad** (long *Matrix, int *Row1, int *Row2, int *Col1, int *Col2, long Template[4])
- int **sfGetOnes** (long *Matrix, int *Pos, int *Neg, int *Eqn, long Template[4])
- void **sfAdd1Real** (long *Element, spREAL *Real)
- void **sfAdd1Imag** (long *Element, spREAL *Imag)
- void **sfAdd1Complex** (long *Element, spREAL *Real, spREAL *Imag)
- void **sfAdd4Real** (long Template[4], spREAL *Real)
- void **sfAdd4Imag** (long Template[4], spREAL *Imag)
- void **sfAdd4Complex** (long Template[4], spREAL *Real, spREAL *Imag)
- int **sfOrderAndFactor** (long *Matrix, spREAL RHS[], spREAL *RelThreshold, spREAL *AbsThreshold, long *DiagPivoting)
- int **sfFactor** (long *Matrix)
- void **sfPartition** (long *Matrix, int *Mode)
- void **sfSolve** (long *Matrix, spREAL RHS[], spREAL Solution[])
- void **sfSolveTransposed** (long *Matrix, spREAL RHS[], spREAL Solution[])
- void **sfPrint** (long *Matrix, long *PrintReordered, long *Data, long *Header)
- long **sfFileMatrix** (long *Matrix, long *Reordered, long *Data, long *Header)
- int **sfFileVector** (long *Matrix, spREAL RHS[])
- int **sfFileStats** (long *Matrix)
- void **sfMNA_Preorder** (long *Matrix)
- void **sfScale** (long *Matrix, spREAL RHS_ScaleFactors[], spREAL SolutionScaleFactors[])
- void **sfMultiply** (long *Matrix, spREAL RHS[], spREAL Solution[])
- void **sfMultTransposed** (long *Matrix, spREAL RHS[], spREAL Solution[])
- void **sfDeterminant** (long *Matrix, spREAL *pDeterminant, spREAL *piDeterminant, int *pExponent)
- int **sfErrorState** (long *Matrix)
- void **sfErrorMessage** (long *Matrix)
- void **sfWhereSingular** (long *Matrix, int *Row, int *Col)
- int **sfGetSize** (long *Matrix, long *External)
- void **sfSetReal** (long *Matrix)
- void **sfSetComplex** (long *Matrix)
- int **sfFillinCount** (long *Matrix)
- int **sfElementCount** (long *Matrix)
- void **sfDeleteRowAndCol** (long *Matrix, int *Row, int *Col)
- spREAL **sfPseudoCondition** (long *Matrix)
- spREAL **sfCondition** (long *Matrix, spREAL *NormOfMatrix, int *pError)
- spREAL **sfNorm** (long *Matrix)
- spREAL **sfLargestElement** (long *Matrix)
- spREAL **sfRoundoff** (long *Matrix, spREAL *Rho)

4.5.1 Detailed Description

This module contains routines that interface Sparse1.4 to a calling program written in fortran. Almost every externally available Sparse1.4 routine has a counterpart defined in this file, with the name the same except the *sp* prefix is changed to *sf*. The *spADD_ELEMENT* and *spADD_QUAD* macros are also replaced with the *sfAdd1* and *sfAdd4* functions defined in this file.

To ease porting this file to different operating systems, the names of the functions can be easily redefined (search for 'Routine Renaming'). A simple example of a FORTRAN program that calls Sparse is included in this file (search for Example). When interfacing to a FORTRAN program, the ARRAY_OFFSET option should be set to NO (see **spConfig.h**).

DISCLAIMER: These interface routines were written by a C programmer who has little experience with FORTRAN. The routines have had minimal testing. Any interface between two languages is going to have portability problems, this one is no exception.

4.5.2 Function Documentation

4.5.2.1 void sfAdd1Complex (long * *Element*, spREAL * *Real*, spREAL * *Imag*)

Adds a complex value to a matrix element. These elements are referenced by pointer, and so must already have been created by **spGetElement**() (p. 12).

Parameters:

Element [INTEGER] Pointer to the element that is to be added to.

Real [REAL or DOUBLE PRECISION] Real portion of the number to be added to the element.

Imag [REAL or DOUBLE PRECISION] Imaginary portion of the number to be added to the element.

4.5.2.2 void sfAdd1Imag (long * *Element*, spREAL * *Imag*)

Adds an imaginary value to a matrix element. These elements are referenced by pointer, and so must already have been created by **spGetElement**() (p. 12).

Parameters:

Element [INTEGER] Pointer to the element that is to be added to.

Imag [REAL or DOUBLE PRECISION] Imaginary portion of the number to be added to the element.

4.5.2.3 void sfAdd1Real (long * *Element*, spREAL * *Real*)

Adds a real value to a matrix element. These elements are referenced by pointer, and so must already have been created by **spGetElement**() (p. 12).

Parameters:

Element [INTEGER] Pointer to the element that is to be added to.

Real [REAL or DOUBLE PRECISION] Real portion of the number to be added to the element.

4.5.2.4 void sfAdd4Complex (long *Template*[4], spREAL * *Real*, spREAL * *Imag*)

Adds a complex value to a set of four elements in a matrix. These elements are referenced by pointer, and so must already have been created by **spGetAdmittance**() (p. 11), **spGetQuad**() (p. 13), or **spGetOnes**() (p. 12).

Parameters:

Template [4]) [INTEGER (4)] Pointer to the element that is to be added to.

Real [REAL or DOUBLE PRECISION] Real portion of the number to be added to the element.

Imag [REAL or DOUBLE PRECISION] Imaginary portion of the number to be added to the element.

4.5.2.5 void sfAdd4Imag (long *Template*[4], spREAL * *Imag*)

Adds an imaginary value to a set of four elements in a matrix. These elements are referenced by pointer, and so must already have been created by **spGetAdmittance**() (p. 11), **spGetQuad**() (p. 13), or **spGetOnes**() (p. 12).

Parameters:

Template [4]) [INTEGER (4)] Pointer to the element that is to be added to.

Imag [REAL or DOUBLE PRECISION] Imaginary portion of the number to be added to the element.

4.5.2.6 void sfAdd4Real (long *Template*[4], spREAL * *Real*)

Adds a real value to a set of four elements in a matrix. These elements are referenced by pointer, and so must already have been created by **spGetAdmittance**() (p. 11), **spGetQuad**() (p. 13), or **spGetOnes**() (p. 12).

Parameters:

Template [4]) [INTEGER (4)] Pointer to the element that is to be added to.

Real [REAL or DOUBLE PRECISION] Real portion of the number to be added to the element.

4.5.2.7 void sfClear (long * *Matrix*)

Sets every element of the matrix to zero and clears the error flag.

Parameters:

Matrix [INTEGER] Pointer to matrix that is to be cleared.

4.5.2.8 spREAL sfCondition (long * *Matrix*, spREAL * *NormOfMatrix*, int * *pError*)

Computes an estimate of the condition number using a variation on the LINPACK condition number estimation algorithm. This quantity is an indicator of ill-conditioning in the matrix. To avoid problems with overflow, the reciprocal of the condition number is returned. If this number is small, and if the matrix is scaled such that uncertainties in the RHS and the matrix entries are equilibrated, then the

matrix is ill-conditioned. If the this number is near one, the matrix is well conditioned. This routine must only be used after a matrix has been factored by **sfOrderAndFactor**() (p. 38) or **sfFactor**() (p. 32) and before it is cleared by **sfClear**() (p. 29) or **spInitialize**() (p. 14).

Unlike the LINPACK condition number estimator, this routines returns the L infinity condition number. This is an artifact of Sparse placing ones on the diagonal of the upper triangular matrix rather than the lower. This difference should be of no importance.

References: A.K. Cline, C.B. Moler, G.W. Stewart, J.H. Wilkinson. An estimate for the condition number of a matrix. SIAM Journal on Numerical Analysis. Vol. 16, No. 2, pages 368-375, April 1979.

J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart. LINPACK User's Guide. SIAM, 1979.

Roger G. Grimes, John G. Lewis. Condition number estimation for sparse matrices. SIAM Journal on Scientific and Statistical Computing. Vol. 2, No. 4, pages 384-388, December 1981.

Dianne Prost O'Leary. Estimating matrix condition numbers. SIAM Journal on Scientific and Statistical Computing. Vol. 1, No. 2, pages 205-209, June 1980.

Returns :

[REAL or DOUBLE PRECISION] The reciprocal of the condition number. If the matrix was singular, zero is returned.

Parameters:

Matrix [INTEGER] Pointer to the matrix.

NormOfMatrix [REAL or DOUBLE PRECISION] The L-infinity norm of the unfactored matrix as computed by **spNorm**() (p. 74).

pError [INTEGER or INTEGER*2] Used to return error code. Possible errors include *spSINGULAR* and *spNO_MEMORY*.

4.5.2.9 long sfCreate (int * *Size*, int * *Complex*, int * *Error*)

Allocates and initializes the data structures associated with a matrix.

Returns :

[INTEGER] A pointer to the matrix is returned cast into an integer. This pointer is then passed and used by the other matrix routines to refer to a particular matrix. If an error occurs, the NULL pointer is returned.

Parameters:

Size [INTEGER] Size of matrix or estimate of size of matrix if matrix is *EXPANDABLE*.

Complex [INTEGER or INTEGER*2] Type of matrix. If *Complex* is 0 then the matrix is real, otherwise the matrix will be complex. Note that if a matrix will be both real and complex, it must be specified here as being complex.

Error [INTEGER or INTEGER*2] Returns error flag, needed because function **spError** (p. 48) will not work correctly if **spCreate**() (p. 8) returns NULL. Possible errors include *spNO_MEMORY*.

4.5.2.10 void sfDeleteRowAndCol (long * *Matrix*, int * *Row*, int * *Col*)

Deletes a row and a column from a matrix.

Sparse will abort if an attempt is made to delete a row or column that doesn't exist.

Parameters:

Matrix [INTEGER] Pointer to the matrix in which the row and column are to be deleted.

Row [INTEGER or INTEGER*2] Row to be deleted.

Col [INTEGER or INTEGER*2] Column to be deleted.

4.5.2.11 void sfDestroy (long * *Matrix*)

Deallocates pointers and elements of matrix.

Parameters:

Matrix [INTEGER] Pointer to the matrix frame which is to be removed from memory.

4.5.2.12 void sfDeterminant (long * *Matrix*, spREAL * *pDeterminant*, spREAL * *piDeterminant*, int * *pExponent*)

This routine in capable of calculating the determinant of the matrix once the LU factorization has been performed. Hence, only use this routine after **spFactor**() (p. 23) and before **spClear**() (p. 11). The determinant equals the product of all the diagonal elements of the lower triangular matrix L, except that this product may need negating. Whether the product or the negative product equals the determinant is determined by the number of row and column interchanges performed. Note that the determinants of matrices can be very large or very small. On large matrices, the determinant can be far larger or smaller than can be represented by a floating point number. For this reason the determinant is scaled to a reasonable value and the logarithm of the scale factor is returned.

Parameters:

Matrix [INTEGER] A pointer to the matrix for which the determinant is desired.

pExponent [INTEGER or INTEGER*2] The logarithm base 10 of the scale factor for the determinant. To find the actual determinant, Exponent should be added to the exponent of DeterminantReal.

pDeterminant [REAL or DOUBLE PRECISION] The real portion of the determinant. This number is scaled to be greater than or equal to 1.0 and less than 10.0.

piDeterminant [REAL or DOUBLE PRECISION] The imaginary portion of the determinant. When the matrix is real this pointer need not be supplied, nothing will be returned. This number is scaled to be greater than or equal to 1.0 and less than 10.0.

4.5.2.13 int sfElementCount (long * *Matrix*)

Returns the total number of total elements in the matrix.

>>> Arguments: Matrix [INTEGER] Pointer to matrix.

4.5.2.14 void sfErrorMessage (long * *Matrix*)

This function prints a Sparse error message to stderr.

Parameters:

Matrix [INTEGER] The matrix for which the error message is desired.

4.5.2.15 int sfErrorState (long * *Matrix*)

This function is used to determine the error status of the given matrix.

Returns :
[INTEGER or INTEGER*2] The error status of the given matrix.

Parameters:
Matrix [INTEGER] The matrix for which the error status is desired.

4.5.2.16 int sfFactor (long * *Matrix*)

This routine is the companion routine to **spOrderAndFactor**() (p. 24). Unlike **sfOrderAndFactor**() (p. 38), **sfFactor**() (p. 32) cannot change the ordering. It is also faster than **sfOrderAndFactor**() (p. 38). The standard way of using these two routines is to first use **sfOrderAndFactor**() (p. 38) for the initial factorization. For subsequent factorizations, **sfFactor**() (p. 32) is used if there is some assurance that little growth will occur (say for example, that the matrix is diagonally dominant). If **sfFactor**() (p. 32) is called for the initial factorization of the matrix, then **sfOrderAndFactor**() (p. 38) is automatically called with the default threshold. This routine uses "row at a time" LU factorization. Pivots are associated with the lower triangular matrix and the diagonals of the upper triangular matrix are ones.

Returns :
[INTEGER or INTEGER*2] The error code is returned. Possible errors include *spNO_MEMORY spSINGULAR spZERO_DIAG spSMALL_PIVOT* Error is cleared in this function.

Parameters:
Matrix [INTEGER] Pointer to matrix.

4.5.2.17 long sfFileMatrix (long * *Matrix*, long * *Reordered*, long * *Data*, long * *Header*)

Writes matrix to file in format suitable to be read back in by the matrix test program. Data is sent to a file with a fixed name (MATRIX_FILE_NAME) because it is impossible to pass strings from FORTRAN to C in a manner that is portable.

Returns :
One is returned if routine was successful, otherwise zero is returned. The calling function can query errno (the system global error variable) as to the reason why this routine failed.

Parameters:
Matrix [INTEGER] Pointer to matrix.

Reordered [LOGICAL] Specifies whether matrix should be output in reordered form, or in original order.

Data [LOGICAL] Indicates that the element values should be output along with the indices for each element. This parameter must be true if matrix is to be read by the sparse test program.

Header [LOGICAL] Indicates that header is desired. This parameter must be true if matrix is to be read by the sparse test program.

4.5.2.18 int sfFileStats (long * *Matrix*)

Writes useful information concerning the matrix to a file. Should be executed after the matrix is factored. Data is sent to a file with a fixed name (STATS_FILE_NAME) because it is impossible to pass strings from FORTRAN to C in a manner that is portable.

Returns :

[LOGICAL] One is returned if routine was successful, otherwise zero is returned. The calling function can query errno (the system global error variable) as to the reason why this routine failed.

Parameters:

Matrix [INTEGER] Pointer to matrix.

4.5.2.19 int sfFileVector (long * *Matrix*, spREAL *RHS*[])

Writes vector to file in format suitable to be read back in by the matrix test program. This routine should be executed after the function sfFileMatrix.

Returns :

One is returned if routine was successful, otherwise zero is returned. The calling function can query errno (the system global error variable) as to the reason why this routine failed.

Parameters:

Matrix [INTEGER] Pointer to matrix.

RHS [REAL (1) or DOUBLE PRECISION (1)] Right-hand side vector. This is only the real portion if *spSEPARATED_COMPLEX_VECTORS* is true.

iRHS [REAL (1) or DOUBLE PRECISION (1)] Right-hand side vector, imaginary portion. Not necessary if matrix is real or if *spSEPARATED_COMPLEX_VECTORS* is set false.

4.5.2.20 int sfFillinCount (long * *Matrix*)

Returns the number of fill-ins in the matrix.

>>> Arguments: Matrix [INTEGER] Pointer to matrix.

4.5.2.21 int sfGetAdmittance (long * *Matrix*, int * *Node1*, int * *Node2*, long *Template*[4])

Performs same function as **sfGetElement**() (p. 34) except rather than one element, all four Matrix elements for a floating component are added. This routine also works if component is grounded. Positive elements are placed at [Node1,Node2] and [Node2,Node1]. This routine is only to be used after **sfCreate**() (p. 30) and before **sfMNA_Preorder**() (p. 36), **sfFactor**() (p. 32) or **sfOrderAndFactor**() (p. 38).

Returns :

[INTEGER or INTEGER*2] The error code. Possible errors include *spNO_MEMORY*. Error is not cleared in this routine.

Parameters:

Matrix [INTEGER] Pointer to the matrix that component is to be entered in.

Node1 [INTEGER or INTEGER*2] Row and column indices for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Node zero is the ground node. In no case may *Node1* be less than zero.

Node2 [INTEGER or INTEGER*2] Row and column indices for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Node zero is the ground node. In no case may *Node2* be less than zero.

Template [INTEGER (4)] Collection of pointers to four elements that are later used to directly address elements. User must supply the template, this routine will fill it.

4.5.2.22 long sfGetElement (long * *Matrix*, int * *Row*, int * *Col*)

Finds element [Row,Col] and returns a pointer to it. If element is not found then it is created and spliced into matrix. This routine is only to be used after **spCreate**() (p. 8) and before **spMNA_Preorder**() (p. 72), **spFactor**() (p. 23) or **spOrderAndFactor**() (p. 24). Returns a pointer to the Real portion of a matrix element. This pointer is later used by sfAddxxxxx() to directly access element.

Returns :

[INTEGER] Returns a pointer to the element. This pointer is then used to directly access the element during successive builds. Returns NULL if *spNO_MEMORY* error occurs. Error is not cleared in this routine.

Parameters:

Matrix [INTEGER] Pointer to the matrix that the element is to be added to.

Row [INTEGER or INTEGER*2] Row index for element. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Elements placed in row zero are discarded. In no case may *Row* be less than zero.

Col [INTEGER or INTEGER*2] Column index for element. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Elements placed in column zero are discarded. In no case may *Col* be less than zero.

4.5.2.23 int sfGetOnes (long * *Matrix*, int * *Pos*, int * *Neg*, int * *Eqn*, long *Template*[4])

Performs similar function to **sfGetQuad**() (p. 35) except this routine is meant for components that do not have an admittance representation.

The following stamp is used:

```
       Pos  Neg  Eqn
Pos  [  .    .    1  ]
Neg  [  .    .   -1  ]
Eqn  [  1   -1    .  ]
```

Returns :

[INTEGER or INTEGER*2] Error code. Possible errors include *spNO_MEMORY*. Error is not cleared in this routine.

Parameters:

Matrix [INTEGER] Pointer to the matrix that component is to be entered in.

Pos [INTEGER or INTEGER*2] See stamp above. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may *Pos* be less than zero.

Neg [INTEGER or INTEGER*2] See stamp above. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may *Neg* be less than zero.

Eqn [INTEGER or INTEGER*2] See stamp above. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may *Eqn* be less than zero.

Template [4]) [INTEGER (4)] Collection of pointers to four elements that are later used to directly address elements. User must supply the template, this routine will fill it.

4.5.2.24 int sfGetQuad (long * *Matrix*, int * *Row1*, int * *Row2*, int * *Col1*, int * *Col2*, long *Template*[4])

Similar to **sfGetAdmittance**() (p. 33), except that **sfGetAdmittance**() (p. 33) only handles 2-terminal components, whereas **sfGetQuad**() (p. 35) handles simple 4-terminals as well. These 4-terminals are simply generalized 2-terminals with the option of having the sense terminals different from the source and sink terminals. **sfGetQuad**() (p. 35) adds four elements to the matrix. Positive elements occur at Row1,Col1 Row2,Col2 while negative elements occur at Row1,Col2 and Row2,Col1. The routine works fine if any of the rows and columns are zero. This routine is only to be used after **sfCreate**() (p. 30) and before **sfMNA_Preorder**() (p. 36), **sfFactor**() (p. 32) or **sfOrderAndFactor**() (p. 38) unless TRANSLATE is set true.

Returns :

[INTEGER or INTEGER*2] Error code. Possible errors include *spNO_MEMORY*. Error is not cleared in this routine.

Parameters:

Matrix [INTEGER] Pointer to the matrix that component is to be entered in.

Row1 [INTEGER or INTEGER*2] First row index for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may *Row1* be less than zero.

Row2 [INTEGER or INTEGER*2] Second row index for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may *Row2* be less than zero.

Col1 [INTEGER or INTEGER*2] First column index for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground column. In no case may *Col1* be less than zero.

Col2 [INTEGER or INTEGER*2] Second column index for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground column. In no case may *Col2* be less than zero.

Template [INTEGER (4)] Collection of pointers to four elements that are later used to directly address elements. User must supply the template, this routine will fill it.

4.5.2.25 int sfGetSize (long * *Matrix*, long * *External*)

Returns the size of the matrix. Either the internal or external size of the matrix is returned.

Parameters:

Matrix [INTEGER] Pointer to matrix.

External [LOGICAL] If External is set true, the external size , i.e., the value of the largest external row or column number encountered is returned. Otherwise the true size of the matrix is returned. These two sizes may differ if the TRANSLATE option is set true.

4.5.2.26 spREAL sfLargestElement (long * *Matrix*)

spLargestElement() (p. 71) finds the magnitude on the largest element in the matrix. If the matrix has not yet been factored, the largest element is found by direct search. If the matrix is factored, a bound on the largest element in any of the reduced submatrices is computed.

Returns :

[REAL or DOUBLE PRECISION] If matrix is not factored, returns the magnitude of the largest element in the matrix. If the matrix is factored, a bound on the magnitude of the largest element in any of the reduced submatrices is returned.

Parameters:

Matrix [INTEGER] Pointer to the matrix.

See also:

spLargestElement() (p. 71)

4.5.2.27 void sfMNA_Preorder (long * *Matrix*)

This routine massages modified node admittance matrices to remove zeros from the diagonal. It takes advantage of the fact that the row and column associated with a zero diagonal usually have structural ones placed symmetricly. This routine should be used only on modified node admittance matrices and should be executed after the matrix has been built but before the factorization begins. It should be executed for the initial factorization only and should be executed before the rows have been linked. Thus it should be run before using **spScale**() (p. 75), **spMultiply**() (p. 73), **spDeleteRowAndCol**() (p. 70), or **spNorm**() (p. 74).

This routine exploits the fact that the structural one are placed in the matrix in symmetric twins. For example, the stamps for grounded and a floating voltage sources are

```
grounded:           floating:
[  x   x   1 ]      [  x   x   1 ]
[  x   x     ]      [  x   x  -1 ]
[  1         ]      [  1  -1     ]
```

Notice for the grounded source, there is one set of twins, and for the grounded, there are two sets. We remove the zero from the diagonal by swapping the rows associated with a set of twins. For example: grounded: floating 1: floating 2:

```
[  1         ]      [  1  -1     ]      [  x   x   1 ]
[  x   x     ]      [  x   x  -1 ]      [  1  -1     ]
[  x   x   1 ]      [  x   x   1 ]      [  x   x  -1 ]
```

It is important to deal with any zero diagonals that only have one set of twins before dealing with those that have more than one because swapping row destroys the symmetry of any twins in the rows being swapped, which may limit future moves. Consider

```
[  x   x   1     ]
[  x   x  -1   1 ]
[  1  -1         ]
[      1         ]
```

There is one set of twins for diagonal 4 and two for diagonal3. Dealing with diagonal for first requires swapping rows 2 and 4.

```
[  x   x   1     ]
[      1         ]
[  1  -1         ]
[  x   x  -1   1 ]
```

We can now deal with diagonal 3 by swapping rows 1 and 3.

```
[  1  -1         ]
[      1         ]
[  x   x   1     ]
[  x   x  -1   1 ]
```

And we are done, there are no zeros left on the diagonal. However, if we originally dealt with diagonal 3 first, we could swap rows 2 and 3

```
[  x   x   1     ]
[  1  -1         ]
[  x   x  -1   1 ]
[      1         ]
```

Diagonal 4 no longer has a symmetric twin and we cannot continue.

So we always take care of lone twins first. When none remain, we choose arbitrarily a set of twins for a diagonal with more than one set and swap the rows corresponding to that twin. We then deal with any lone twins that were created and repeat the procedure until no zero diagonals with symmetric twins remain.

In this particular implementation, columns are swapped rather than rows. The algorithm used in this function was developed by Ken Kundert and Tom Quarles.

Parameters:

Matrix [INTEGER] Pointer to the matrix to be preordered.

4.5.2.28 void sfMultiply (long * *Matrix*, spREAL *RHS*[], spREAL *Solution*[])

Multiplies matrix by solution vector to find source vector. Assumes matrix has not been factored. This routine can be used as a test to see if solutions are correct. It should not be used before PreorderFo-ModifiedNodal().

Parameters:

Matrix [INTEGER] Pointer to the matrix.

RHS [**REAL**(1) (p. 20) or DOUBLE PRECISION(1)] RHS is the right hand side. This is what is being solved for.

Solution [**REAL**(1) (p. 20) or DOUBLE PRECISION(1)] Solution is the vector being multiplied by the matrix.

iRHS [**REAL**(1) (p. 20) or DOUBLE PRECISION(1)] iRHS is the imaginary portion of the right hand side. This is what is being solved for. This is only necessary if the matrix is complex and spSEPARATED_COMPLEX_VECTORS is true.

iSolution [**REAL**(1) (p. 20) or DOUBLE PRECISION(1)] iSolution is the imaginary portion of the vector being multiplied by the matrix. This is only necessary if the matrix is complex and spSEPARATED_COMPLEX_VECTORS is true.

4.5.2.29 void sfMultTransposed (long * *Matrix*, spREAL *RHS*[ ], spREAL *Solution*[ ])

Multiplies transposed matrix by solution vector to find source vector. Assumes matrix has not been factored. This routine can be used as a test to see if solutions are correct. It should not be used before PreorderFoModifiedNodal().

Parameters:

Matrix [INTEGER] Pointer to the matrix.

RHS [**REAL**(1) (p. 20) or DOUBLE PRECISION(1)] RHS is the right hand side. This is what is being solved for.

Solution [**REAL**(1) (p. 20) or DOUBLE PRECISION(1)] Solution is the vector being multiplied by the matrix.

iRHS [**REAL**(1) (p. 20) or DOUBLE PRECISION(1)] iRHS is the imaginary portion of the right hand side. This is what is being solved for. This is only necessary if the matrix is complex and spSEPARATED_COMPLEX_VECTORS is true.

iSolution [**REAL**(1) (p. 20) or DOUBLE PRECISION(1)] iSolution is the imaginary portion of the vector being multiplied by the matrix. This is only necessary if the matrix is complex and spSEPARATED_COMPLEX_VECTORS is true.

4.5.2.30 spREAL sfNorm (long * *Matrix*)

Computes the L-infinity norm of an unfactored matrix. It is a fatal error to pass this routine a factored matrix.

Returns :

[REAL or DOUBLE PRECISION] The largest absolute row sum of matrix.

Parameters:

Matrix [INTEGER] Pointer to the matrix.

4.5.2.31 int sfOrderAndFactor (long * *Matrix*, spREAL *RHS*[ ], spREAL * *RelThreshold*, spREAL * *AbsThreshold*, long * *DiagPivoting*)

This routine chooses a pivot order for the matrix and factors it into LU form. It handles both the initial factorization and subsequent factorizations when a reordering is desired. This is handled in a manner that is transparent to the user. The routine uses a variation of Gauss's method where the pivots are associated with L and the diagonal terms of U are one.

Returns :

[INTEGER of INTEGER*2] The error code is returned. Possible errors include *spNO_MEMORY*, *spSINGULAR*, and *spSMALL_PIVOT*. Error is cleared in this function.

Returns :

Matrix [INTEGER] Pointer to matrix. RHS [REAL (1) or DOUBLE PRECISION (1)] Representative right-hand side vector that is used to determine pivoting order when the right hand side vector is sparse. If *RHS* is a NULL pointer then the RHS vector is assumed to be full and it is not used when determining the pivoting order. RelThreshold [REAL or DOUBLE PRECISION] This number determines what the pivot relative threshold will be. It should be between zero and one. If it is one then the pivoting method becomes complete pivoting, which is very slow and tends to fill up the matrix. If it is set close to zero the pivoting method becomes strict Markowitz with no threshold. The pivot threshold is used to eliminate pivot candidates that would cause excessive element growth if they were used. Element growth is the cause of roundoff error. Element growth occurs even in well-conditioned matrices. Setting the *RelThreshold* large will reduce element growth and roundoff error, but setting it too large will cause execution time to be excessive and will result in a large number of fill-ins. If this occurs, accuracy can actually be degraded because of the large number of operations required on the matrix due to the large number of fill-ins. A good value seems to be 0.001. The default is chosen by giving a value larger than one or less than or equal to zero. This value should be increased and the matrix resolved if growth is found to be excessive. Changing the pivot threshold does not improve performance on matrices where growth is low, as is often the case with ill-conditioned matrices. Once a valid threshold is given, it becomes the new default. The default value of RelThreshold was chosen for use with nearly diagonally dominant matrices such as node- and modified-node admittance matrices. For these matrices it is usually best to use diagonal pivoting. For matrices without a strong diagonal, it is usually best to use a larger threshold, such as 0.01 or 0.1. AbsThreshold [REAL or DOUBLE PRECISION] The absolute magnitude an element must have to be considered as a pivot candidate, except as a last resort. This number should be set significantly smaller than the smallest diagonal element that is is expected to be placed in the matrix. If there is no reasonable prediction for the lower bound on these elements, then *AbsThreshold* should be set to zero. *AbsThreshold* is used to reduce the possibility of choosing as a pivot an element that has suffered heavy cancellation and as a result mainly consists of roundoff error. Once a valid threshold is given, it becomes the new default. DiagPivoting [LOGICAL] A flag indicating that pivot selection should be confined to the diagonal if possible. If DiagPivoting is nonzero and if *DIAGONAL_PIVOTING* is enabled pivots will be chosen only from the diagonal unless there are no diagonal elements that satisfy the threshold criteria. Otherwise, the entire reduced submatrix is searched when looking for a pivot. The diagonal pivoting in Sparse is efficient and well refined, while the off-diagonal pivoting is not. For symmetric and near symmetric matrices, it is best to use diagonal pivoting because it results in the best performance when reordering the matrix and when factoring the matrix without ordering. If there is a considerable amount of nonsymmetry in the matrix, then off-diagonal pivoting may result in a better equation ordering simply because there are more pivot candidates to choose from. A better ordering results in faster subsequent factorizations. However, the initial pivot selection process takes considerably longer for off-diagonal pivoting.

4.5.2.32 void sfPartition (long * *Matrix*, int * *Mode*)

This routine determines the cost to factor each row using both direct and indirect addressing and decides, on a row-by-row basis, which addressing mode is fastest. This information is used in **sfFactor**() (p. 32) to speed the factorization.

When factoring a previously ordered matrix using **sfFactor**() (p. 32), Sparse operates on a row-at-a-time basis. For speed, on each step, the row being updated is copied into a full vector and the operations are performed on that vector. This can be done one of two ways, either using direct addressing or indirect addressing. Direct addressing is fastest when the matrix is relatively dense and indirect addressing is best when the matrix is quite sparse. The user selects the type of partition used with *Mode*. If *Mode* is set to *spDIRECT_PARTITION*, then the all rows are placed in the direct addressing partition. Similarly, if *Mode* is set to *spINDIRECT_PARTITION*, then the all rows are placed

in the indirect addressing partition. By setting *Mode* to *spAUTO_PARTITION*, the user allows *Sparse* to select the partition for each row individually. **sfFactor**() (p. 32) generally runs faster if Sparse is allowed to choose its own partitioning, however choosing a partition is expensive. The time required to choose a partition is of the same order of the cost to factor the matrix. If you plan to factor a large number of matrices with the same structure, it is best to let *Sparse* choose the partition. Otherwise, you should choose the partition based on the predicted density of the matrix.

Parameters:

Matrix [INTEGER] Pointer to matrix.

Mode [INTEGER or INTEGER*2] Mode must be one of three special codes: *spDIRECT_PARTITION*, *spINDIRECT_PARTITION*, or *spAUTO_PARTITION*.

4.5.2.33 void sfPrint (long * *Matrix*, long * *PrintReordered*, long * *Data*, long * *Header*)

Formats and send the matrix to standard output. Some elementary statistics are also output. The matrix is output in a format that is readable by people.

Parameters:

Matrix [INTEGER] Pointer to matrix.

PrintReordered [LOGICAL] Indicates whether the matrix should be printed out in its original form, as input by the user, or whether it should be printed in its reordered form, as used by the matrix routines. A zero indicates that the matrix should be printed as inputed, a one indicates that it should be printed reordered.

Data [LOGICAL] Boolean flag that when false indicates that output should be compressed such that only the existence of an element should be indicated rather than giving the actual value. Thus 10 times as many can be printed on a row. A zero signifies that the matrix should be printed compressed. A one indicates that the matrix should be printed in all its glory.

Header [LOGICAL] Flag indicating that extra information such as the row and column numbers should be printed.

4.5.2.34 spREAL sfPseudoCondition (long * *Matrix*)

Computes the magnitude of the ratio of the largest to the smallest pivots. This quantity is an indicator of ill-conditioning in the matrix. If this ratio is large, and if the matrix is scaled such that uncertainties in the RHS and the matrix entries are equilibrated, then the matrix is ill-conditioned. However, a small ratio does not necessarily imply that the matrix is well-conditioned. This routine must only be used after a matrix has been factored by **sfOrderAndFactor**() (p. 38) or **sfFactor**() (p. 32) and before it is cleared by **sfClear**() (p. 29) or **spInitialize**() (p. 14). The pseudocondition is faster to compute than the condition number calculated by **sfCondition**() (p. 29), but is not as informative.

Returns :

[REAL or DOUBLE PRECISION] The magnitude of the ratio of the largest to smallest pivot used during previous factorization. If the matrix was singular, zero is returned.

Parameters:

Matrix [INTEGER] Pointer to the matrix.

4.5.2.35 spREAL sfRoundoff (long * *Matrix*, spREAL * *Rho*)

This routine, along with **spLargestElement**() (p. 71), are used to gauge the stability of a factorization. See description of **spLargestElement**() (p. 71) for more information.

Returns :

[REAL or DOUBLE PRECISION] Returns a bound on the magnitude of the largest element in E = A - LU.

Parameters:

Matrix [INTEGER] Pointer to the matrix.

Rho [REAL or DOUBLE PRECISION] The bound on the magnitude of the largest element in any of the reduced submatrices. This is the number computed by the function **spLargestElement**() (p. 71) when given a factored matrix. If this number is negative, the bound will be computed automatically.

See also:

spRoundoff() (p. 75)

4.5.2.36 void sfScale (long * *Matrix*, spREAL *RHS_ScaleFactors*[ ], spREAL *SolutionScaleFactors*[ ])

This function scales the matrix to enhance the possibility of finding a good pivoting order. Note that scaling enhances accuracy of the solution only if it affects the pivoting order, so it makes no sense to scale the matrix before **spFactor**() (p. 23). If scaling is desired it should be done before **spOrderAndFactor**() (p. 24). There are several things to take into account when choosing the scale factors. First, the scale factors are directly multiplied against the elements in the matrix. To prevent roundoff, each scale factor should be equal to an integer power of the number base of the machine. Since most machines operate in base two, scale factors should be a power of two. Second, the matrix should be scaled such that the matrix of element uncertainties is equilibrated. Third, this function multiplies the scale factors by the elements, so if one row tends to have uncertainties 1000 times smaller than the other rows, then its scale factor should be 1024, not 1/1024. Fourth, to save time, this function does not scale rows or columns if their scale factors are equal to one. Thus, the scale factors should be normalized to the most common scale factor. Rows and columns should be normalized separately. For example, if the size of the matrix is 100 and 10 rows tend to have uncertainties near 1e-6 and the remaining 90 have uncertainties near 1e-12, then the scale factor for the 10 should be 1/1,048,576 and the scale factors for the remaining 90 should be 1. Fifth, since this routine directly operates on the matrix, it is necessary to apply the scale factors to the RHS and Solution vectors. It may be easier to simply use **spOrderAndFactor**() (p. 24) on a scaled matrix to choose the pivoting order, and then throw away the matrix. Subsequent factorizations, performed with **spFactor**() (p. 23), will not need to have the RHS and Solution vectors descaled. Lastly, this function should not be executed before the function spMNA_Preorder.

Parameters:

Matrix [INTEGER] Pointer to the matrix to be scaled.

SolutionScaleFactors [**REAL**(1) (p. 20) or DOUBLE PRECISION(1)] The array of Solution scale factors. These factors scale the columns. All scale factors are real valued.

RHS_ScaleFactors [**REAL**(1) (p. 20) or DOUBLE PRECISION(1)] The array of RHS scale factors. These factors scale the rows. All scale factors are real valued.

4.5.2.37 void sfSetComplex (long * *Matrix*)

Forces matrix to be complex.

Parameters:

Matrix [INTEGER] Pointer to matrix.

4.5.2.38 void sfSetReal (long * *Matrix*)

Forces matrix to be real.

Parameters:

Matrix [INTEGER] Pointer to matrix.

4.5.2.39 void sfSolve (long * *Matrix*, spREAL *RHS*[], spREAL *Solution*[])

Performs forward elimination and back substitution to find the unknown vector from the RHS vector and factored matrix. This routine assumes that the pivots are associated with the lower triangular (L) matrix and that the diagonal of the upper triangular (U) matrix consists of ones. This routine arranges the computation in different way than is traditionally used in order to exploit the sparsity of the right-hand side. See the reference in spRevision.

Parameters:

Matrix [INTEGER] Pointer to matrix.

RHS [REAL (1) or DOUBLE PRECISION (1)] *RHS* is the input data array, the right hand side. This data is undisturbed and may be reused for other solves.

Solution [REAL (1) or DOUBLE PRECISION (1)] *Solution* is the output data array. This routine is constructed such that *RHS* and *Solution* can be the same array.

iRHS [REAL (1) or DOUBLE PRECISION (1)] *iRHS* is the imaginary portion of the input data array, the right hand side. This data is undisturbed and may be reused for other solves. This argument is only necessary if matrix is complex and if *spSEPARATED_COMPLEX_VECTOR* is set true.

iSolution [REAL (1) or DOUBLE PRECISION (1)] *iSolution* is the imaginary portion of the output data array. This routine is constructed such that *iRHS* and *iSolution* can be the same array. This argument is only necessary if matrix is complex and if *spSEPARATED_COMPLEX_VECTOR* is set true.

4.5.2.40 void sfSolveTransposed (long * *Matrix*, spREAL *RHS*[], spREAL *Solution*[])

Performs forward elimination and back substitution to find the unknown vector from the RHS vector and transposed factored matrix. This routine is useful when performing sensitivity analysis on a circuit using the adjoint method. This routine assumes that the pivots are associated with the untransposed lower triangular (L) matrix and that the diagonal of the untransposed upper triangular (U) matrix consists of ones.

Parameters:

Matrix [INTEGER] Pointer to matrix.

RHS [REAL (1) or DOUBLE PRECISION (1)] *RHS* is the input data array, the right hand side. This data is undisturbed and may be reused for other solves.

Solution [REAL (1) or DOUBLE PRECISION (1)] *Solution* is the output data array. This routine is constructed such that *RHS* and *Solution* can be the same array.

iRHS [REAL (1) or DOUBLE PRECISION (1)] *iRHS* is the imaginary portion of the input data array, the right hand side. This data is undisturbed and may be reused for other solves. If *spSEPARATED_COMPLEX_VECTOR* is set false, or if matrix is real, there is no need to supply this array.

iSolution [REAL (1) or DOUBLE PRECISION (1)] *iSolution* is the imaginary portion of the output data array. This routine is constructed such that *iRHS* and *iSolution* can be the same array. If *spSEPARATED_COMPLEX_VECTOR* is set false, or if matrix is real, there is no need to supply this array.

4.5.2.41 void sfStripFills (long * *Matrix*)

Strips the matrix of all fill-ins.

Parameters:

Matrix [INTEGER] Pointer to the matrix to be stripped.

4.5.2.42 void sfWhereSingular (long * *Matrix*, int * *Row*, int * *Col*)

This function returns the row and column number where the matrix was detected as singular or where a zero was detected on the diagonal.

Parameters:

Matrix [INTEGER] The matrix for which the error status is desired.

pRow [INTEGER or INTEGER*2] The row number.

pCol [INTEGER or INTEGER*2] The column number.

4.6 spMatrix.h File Reference

```
#include "spConfig.h"
```

Compounds

- struct **spTemplate**

Defines

- #define **spOKAY** 0
- #define **spSMALL_PIVOT** 1
- #define **spZERO_DIAG** 2
- #define **spSINGULAR** 3
- #define **spMANGLED** 4
- #define **spNO_MEMORY** 5
- #define **spPANIC** 6
- #define **spFATAL** 2
- #define **spREAL** double
- #define **spDEFAULT_PARTITION** 0
- #define **spDIRECT_PARTITION** 1
- #define **spINDIRECT_PARTITION** 2
- #define **spAUTO_PARTITION** 3
- #define **spADD_REAL_ELEMENT**(element, real) *(element) += real
- #define **spADD_IMAG_ELEMENT**(element, imag) *(element+1) += imag
- #define **spADD_COMPLEX_ELEMENT**(element, real, imag)
- #define **spADD_REAL_QUAD**(template, real)
- #define **spADD_IMAG_QUAD**(template, imag)
- #define **spADD_COMPLEX_QUAD**(template, real, imag)

Typedefs

- typedef spGenericPtr **spMatrix**
- typedef spREAL **spElement**
- typedef int **spError**

Functions

- spcEXTERN void **spClear** (**spMatrix**)
- spcEXTERN spREAL **spCondition** (**spMatrix**, spREAL, int *)
- spcEXTERN **spMatrix spCreate** (int, int, **spError** *)
- spcEXTERN void **spDeleteRowAndCol** (**spMatrix**, int, int)
- spcEXTERN void **spDestroy** (**spMatrix**)
- spcEXTERN int **spElementCount** (**spMatrix**)
- spcEXTERN **spError spErrorState** (**spMatrix**)
- spcEXTERN void **spErrorMessage** (**spMatrix**, FILE *, char *)
- spcEXTERN **spError spFactor** (**spMatrix**)
- spcEXTERN int **spFileMatrix** (**spMatrix**, char *, char *, int, int, int)

- spcEXTERN int **spFileStats** (**spMatrix**, char *, char *)
- spcEXTERN int **spFillinCount** (**spMatrix**)
- spcEXTERN **spElement** * **spFindElement** (**spMatrix**, int, int)
- spcEXTERN **spError spGetAdmittance** (**spMatrix**, int, int, struct **spTemplate** *)
- spcEXTERN **spElement** * **spGetElement** (**spMatrix**, int, int)
- spcEXTERN spGenericPtr **spGetInitInfo** (**spElement** *)
- spcEXTERN **spError spGetOnes** (**spMatrix**, int, int, int, struct **spTemplate** *)
- spcEXTERN **spError spGetQuad** (**spMatrix**, int, int, int, int, struct **spTemplate** *)
- spcEXTERN int **spGetSize** (**spMatrix**, int)
- spcEXTERN int **spInitialize** (**spMatrix**, int(*pInit)(**spElement** *, spGenericPtr, int, int))
- spcEXTERN void **spInstallInitInfo** (**spElement** *, spGenericPtr)
- spcEXTERN spREAL **spLargestElement** (**spMatrix**)
- spcEXTERN void **spMNA_Preorder** (**spMatrix**)
- spcEXTERN spREAL **spNorm** (**spMatrix**)
- spcEXTERN **spError spOrderAndFactor** (**spMatrix**, spREAL[], spREAL, spREAL, int)
- spcEXTERN void **spPartition** (**spMatrix**, int)
- spcEXTERN void **spPrint** (**spMatrix**, int, int, int)
- spcEXTERN spREAL **spPseudoCondition** (**spMatrix**)
- spcEXTERN spREAL **spRoundoff** (**spMatrix**, spREAL)
- spcEXTERN void **spScale** (**spMatrix**, spREAL[], spREAL[])
- spcEXTERN void **spSetComplex** (**spMatrix**)
- spcEXTERN void **spSetReal** (**spMatrix**)
- spcEXTERN void **spStripFills** (**spMatrix**)
- spcEXTERN void **spWhereSingular** (**spMatrix**, int *, int *)
- spcEXTERN void **spDeterminant** (**spMatrix**, int *, spREAL *, spREAL *)
- spcEXTERN int **spFileVector** (**spMatrix**, char *, spREAL[])
- spcEXTERN void **spMultiply** (**spMatrix**, spREAL[], spREAL[])
- spcEXTERN void **spMultTransposed** (**spMatrix**, spREAL[], spREAL[])
- spcEXTERN void **spSolve** (**spMatrix**, spREAL[], spREAL[])
- spcEXTERN void **spSolveTransposed** (**spMatrix**, spREAL[], spREAL[])

4.6.1 Detailed Description

This file contains definitions that are useful to the calling program. In particular, this file contains error keyword definitions, some macro functions that are used to quickly enter data into the matrix and the type definition of a data structure that acts as a template for entering admittances into the matrix. Also included is the type definitions for the various functions available to the user.

Objects that begin with the *spc* prefix are considered private and should not be used.

Author:
Kenneth S. Kundert <kundert@users.sourceforge.net>

4.6.2 Define Documentation

4.6.2.1 #define spADD_COMPLEX_ELEMENT(element, real, imag)

Value:

```
{   *(element) += real;                                 \
    *(element+1) += imag;                               \
}
```

Macro function that adds data to a complex element in the matrix by a pointer.

4.6.2.2 #define spADD_COMPLEX_QUAD(template, real, imag)

Value:

```
{   *((template).Element1) += real;             \
    *((template).Element2) += real;             \
    *((template).Element3Negated) -= real;      \
    *((template).Element4Negated) -= real;      \
    *((template).Element1+1) += imag;           \
    *((template).Element2+1) += imag;           \
    *((template).Element3Negated+1) -= imag;    \
    *((template).Element4Negated+1) -= imag;    \
}
```

Macro function that adds data to each of the four complex matrix elements specified by the given template.

4.6.2.3 #define spADD_IMAG_ELEMENT(element, imag) *(element+1) += imag

Macro function that adds data to a imaginary element in the matrix by a pointer.

4.6.2.4 #define spADD_IMAG_QUAD(template, imag)

Value:

```
{   *((template).Element1+1) += imag;           \
    *((template).Element2+1) += imag;           \
    *((template).Element3Negated+1) -= imag;    \
    *((template).Element4Negated+1) -= imag;    \
}
```

Macro function that adds data to each of the four imaginary matrix elements specified by the given template.

4.6.2.5 #define spADD_REAL_ELEMENT(element, real) *(element) += real

Macro function that adds data to a real element in the matrix by a pointer.

4.6.2.6 #define spADD_REAL_QUAD(template, real)

Value:

```
{   *((template).Element1) += real;             \
    *((template).Element2) += real;             \
    *((template).Element3Negated) -= real;      \
    *((template).Element4Negated) -= real;      \
}
```

Macro function that adds data to each of the four real matrix elements specified by the given template.

4.6.2.7 #define spAUTO_PARTITION 3

Partition code for **spPartition**() (p. 60). Indicates that *Sparse* should chose the best partition for each row based on some simple rules. This is generally preferred.

See also:
spPartition() (p. 60)

4.6.2.8 #define spDEFAULT_PARTITION 0

Partition code for **spPartition**() (p. 60). Indicates that the default partitioning mode should be used.

See also:
spPartition() (p. 60)

4.6.2.9 #define spDIRECT_PARTITION 1

Partition code for **spPartition**() (p. 60). Indicates that all rows should be placed in the direct addressing partition.

See also:
spPartition() (p. 60)

4.6.2.10 #define spFATAL 2

Error code that is not an error flag, but rather the dividing line between fatal errors and warnings.

4.6.2.11 #define spINDIRECT_PARTITION 2

Partition code for **spPartition**() (p. 60). Indicates that all rows should be placed in the indirect addressing partition.

See also:
spPartition() (p. 60)

4.6.2.12 #define spMANGLED 4

Fatal error code that indicates that, matrix has been mangled, results of requested operation are garbage.

4.6.2.13 #define spNO_MEMORY 5

Fatal error code that indicates that not enough memory is available.

4.6.2.14 #define spOKAY 0

Error code that indicates that no error has occurred.

4.6.2.15 #define spPANIC 6

Fatal error code that indicates that the routines are not prepared to handle the matrix that has been requested. This may occur when the matrix is specified to be real and the routines are not compiled for real matrices, or when the matrix is specified to be complex and the routines are not compiled to handle complex matrices.

4.6.2.16 #define spREAL double

Defines the precision of the arithmetic used by *Sparse* will use. Double precision is suggested as being most appropriate for circuit simulation and for C. However, it is possible to change spREAL to a float for single precision arithmetic. Note that in C, single precision arithmetic is often slower than double precision. Sparse internally refers to spREALs as RealNumbers.

4.6.2.17 #define spSINGULAR 3

Fatal error code that indicates that, matrix is singular, so no unique solution exists.

4.6.2.18 #define spSMALL_PIVOT 1

Non-fatal error code that indicates that, when reordering the matrix, no element was found that satisfies the absolute threshold criteria. The largest element in the matrix was chosen as pivot.

4.6.2.19 #define spZERO_DIAG 2

Fatal error code that indicates that, a zero was encountered on the diagonal the matrix. This does not necessarily imply that the matrix is singular. When this error occurs, the matrix should be reconstructed and factored using **spOrderAndFactor**() (p. 59).

4.6.3 Typedef Documentation

4.6.3.1 typedef spREAL spElement

Declares the type of the a pointer to a matrix element.

4.6.3.2 typedef int spError

Declares the type of the Sparse error codes.

4.6.3.3 typedef spGenericPtr spMatrix

Declares the type of the a pointer to a matrix.

4.6.4 Function Documentation

4.6.4.1 spcEXTERN void spClear (spMatrix *eMatrix*)

Sets every element of the matrix to zero and clears the error flag.

Parameters:

eMatrix Pointer to matrix that is to be cleared.

4.6.4.2 spcEXTERN spREAL spCondition (spMatrix *eMatrix*, spREAL *NormOfMatrix*, int * *pError*)

Computes an estimate of the condition number using a variation on the LINPACK condition number estimation algorithm. This quantity is an indicator of ill-conditioning in the matrix. To avoid problems with overflow, the reciprocal of the condition number is returned. If this number is small, and if the matrix is scaled such that uncertainties in the RHS and the matrix entries are equilibrated, then the matrix is ill-conditioned. If the this number is near one, the matrix is well conditioned. This routine must only be used after a matrix has been factored by **spOrderAndFactor**() (p. 59) or **spFactor**() (p. 51) and before it is cleared by **spClear**() (p. 49) or **spInitialize**().

Unlike the LINPACK condition number estimator, this routines returns the L infinity condition number. This is an artifact of Sparse placing ones on the diagonal of the upper triangular matrix rather than the lower. This difference should be of no importance.

References:

A.K. Cline, C.B. Moler, G.W. Stewart, J.H. Wilkinson. An estimate for the condition number of a matrix. SIAM Journal on Numerical Analysis. Vol. 16, No. 2, pages 368-375, April 1979.

J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart. LINPACK User's Guide. SIAM, 1979.

Roger G. Grimes, John G. Lewis. Condition number estimation for sparse matrices. SIAM Journal on Scientific and Statistical Computing. Vol. 2, No. 4, pages 384-388, December 1981.

Dianne Prost O'Leary. Estimating matrix condition numbers. SIAM Journal on Scientific and Statistical Computing. Vol. 1, No. 2, pages 205-209, June 1980.

Returns :

The reciprocal of the condition number. If the matrix was singular, zero is returned.

Parameters:

eMatrix Pointer to the matrix.

NormOfMatrix The L-infinity norm of the unfactored matrix as computed by **spNorm**() (p. 59).

pError Used to return error code. Possible errors include *spSINGULAR* or *spNO_MEMORY*.

4.6.4.3 spcEXTERN spMatrix spCreate (int *Size*, int *Complex*, spError * *pError*)

Allocates and initializes the data structures associated with a matrix.

Returns :

A pointer to the matrix is returned cast into **spMatrix** (typically a pointer to a void). This pointer is then passed and used by the other matrix routines to refer to a particular matrix. If an error occurs, the *NULL* pointer is returned.

Parameters:

Size Size of matrix or estimate of size of matrix if matrix is *EXPANDABLE*.

Complex Type of matrix. If *Complex* is 0 then the matrix is real, otherwise the matrix will be complex. Note that if the routines are not set up to handle the type of matrix requested, then an *spPANIC* error will occur. Further note that if a matrix will be both real and complex, it must be specified here as being complex.

pError Returns error flag, needed because function **spErrorState**() (p. 51) will not work correctly if **spCreate**() (p. 49) returns *NULL*. Possible errors include *spNO_MEMORY* and *spPANIC*.

4.6.4.4 spcEXTERN void spDeleteRowAndCol (spMatrix *eMatrix*, int *Row*, int *Col*)

Deletes a row and a column from a matrix.

Sparse will abort if an attempt is made to delete a row or column that doesn't exist.

Parameters:

eMatrix Pointer to the matrix in which the row and column are to be deleted.

Row Row to be deleted.

Col Column to be deleted.

4.6.4.5 spcEXTERN void spDestroy (spMatrix *eMatrix*)

Destroys a matrix and frees all memory associated with it.

Parameters:

eMatrix Pointer to the matrix frame which is to be destroyed.

4.6.4.6 spcEXTERN void spDeterminant (spMatrix *eMatrix*, int * *pExponent*, spREAL * *pDeterminant*, spREAL * *piDeterminant*)

This routine in capable of calculating the determinant of the matrix once the LU factorization has been performed. Hence, only use this routine after **spFactor**() (p. 51) and before **spClear**() (p. 49). The determinant equals the product of all the diagonal elements of the lower triangular matrix L, except that this product may need negating. Whether the product or the negative product equals the determinant is determined by the number of row and column interchanges performed. Note that the determinants of matrices can be very large or very small. On large matrices, the determinant can be far larger or smaller than can be represented by a floating point number. For this reason the determinant is scaled to a reasonable value and the logarithm of the scale factor is returned.

Parameters:

eMatrix A pointer to the matrix for which the determinant is desired.

pExponent The logarithm base 10 of the scale factor for the determinant. To find the actual determinant, Exponent should be added to the exponent of Determinant.

pDeterminant The real portion of the determinant. This number is scaled to be greater than or equal to 1.0 and less than 10.0.

piDeterminant The imaginary portion of the determinant. When the matrix is real this pointer need not be supplied, nothing will be returned. This number is scaled to be greater than or equal to 1.0 and less than 10.0.

4.6.4.7 spcEXTERN int spElementCount (spMatrix *eMatrix*)

This function returns the total number of elements (including fill-ins) that currently exists in a matrix.

Parameters:

eMatrix Pointer to matrix.

4.6.4.8 spcEXTERN void spErrorMessage (spMatrix *eMatrix*, FILE * *Stream*, char * *Originator*)

This routine prints a short message describing the error error state of sparse. No message is produced if there is no error. The error state is cleared.

Parameters:

eMatrix Matrix for which the error message is to be printed.

Stream Stream to which the error message is to be printed.

Originator Name of originator of error message. If NULL, 'sparse' is used. If zero-length string, no originator is printed.

4.6.4.9 spcEXTERN spError spErrorState (spMatrix *eMatrix*)

This function returns the error status of the given matrix.

Returns :

The error status of the given matrix.

Parameters:

eMatrix The pointer to the matrix for which the error status is desired.

4.6.4.10 spcEXTERN spError spFactor (spMatrix *eMatrix*)

This routine is the companion routine to **spOrderAndFactor**() (p. 59). Unlike **spOrderAndFactor**() (p. 59), **spFactor**() (p. 51) cannot change the ordering. It is also faster than **spOrderAndFactor**() (p. 59). The standard way of using these two routines is to first use **spOrderAndFactor**() (p. 59) for the initial factorization. For subsequent factorizations, **spFactor**() (p. 51) is used if there is some assurance that little growth will occur (say for example, that the matrix is diagonally dominant). If **spFactor**() (p. 51) is called for the initial factorization of the matrix, then **spOrderAndFactor**() (p. 59) is automatically called with the default threshold. This routine uses "row at a time" *LU* factorization. Pivots are associated with the lower triangular matrix and the diagonals of the upper triangular matrix are ones.

Returns :

The error code is returned. Possible errors are *spNO_MEMORY*, *spSINGULAR*, *spZERO_DIAG* and *spSMALL_PIVOT*. Error is cleared upon entering this function.

Parameters:

eMatrix Pointer to matrix.

See also:

spOrderAndFactor() (p. 59)

4.6.4.11 spcEXTERN int spFileMatrix (spMatrix *eMatrix*, char * *File*, char * *Label*, int *Reordered*, int *Data*, int *Header*)

Writes matrix to file in format suitable to be read back in by the matrix test program.

Returns :
One is returned if routine was successful, otherwise zero is returned. The calling function can query *errno* (the system global error variable) as to the reason why this routine failed.

Parameters:
eMatrix Pointer to matrix.

File Name of file into which matrix is to be written.

Label String that is transferred to file and is used as a label.

Reordered Specifies whether matrix should be output in reordered form, or in original order.

Data Indicates that the element values should be output along with the indices for each element. This parameter must be true if matrix is to be read by the sparse test program.

Header Indicates that header is desired. This parameter must be true if matrix is to be read by the sparse test program.

4.6.4.12 spcEXTERN int spFileStats (spMatrix *eMatrix*, char * *File*, char * *Label*)

Writes useful information concerning the matrix to a file. Should be executed after the matrix is factored.

Returns :
One is returned if routine was successful, otherwise zero is returned. The calling function can query *errno* (the system global error variable) as to the reason why this routine failed.

Parameters:
eMatrix Pointer to matrix.

File Name of file into which matrix is to be written.

Label String that is transferred to file and is used as a label.

4.6.4.13 spcEXTERN int spFileVector (spMatrix *eMatrix*, char * *File*, spREAL *RHS*[])

Writes vector to file in format suitable to be read back in by the matrix test program. This routine should be executed after the function spFileMatrix.

Returns :
One is returned if routine was successful, otherwise zero is returned. The calling function can query *errno* (the system global error variable) as to the reason why this routine failed.

Parameters:
eMatrix Pointer to matrix.

File Name of file into which matrix is to be written.

RHS Right-hand side vector. This is only the real portion if *spSEPARATED_COMPLEX_VECTORS* is true.

iRHS Right-hand side vector, imaginary portion. Not necessary if matrix is real or if *spSEPARATED_COMPLEX_VECTORS* is set false. *iRHS* is a macro that replaces itself with ', iRHS' if the options *spCOMPLEX* and *spSEPARATED_COMPLEX_VECTORS* are set, otherwise it disappears without a trace.

4.6.4.14 spcEXTERN int spFillinCount (spMatrix *eMatrix*)

This function returns the number of fill-ins that currently exists in a matrix.

Parameters:

eMatrix Pointer to matrix.

4.6.4.15 spcEXTERN spElement* spFindElement (spMatrix *eMatrix*, int *Row*, int *Col*)

This routine is used to find an element given its indices. It will not create it if it does not exist.

Returns :

A pointer to the desired element, or *NULL* if it does not exist.

Parameters:

eMatrix Pointer to matrix.

Row Row index for element.

Col Column index for element.

See also:

spGetElement() (p. 54)

4.6.4.16 spcEXTERN spError spGetAdmittance (spMatrix *Matrix*, int *Node1*, int *Node2*, struct spTemplate * *Template*)

Performs same function as **spGetElement**() (p. 54) except rather than one element, all four matrix elements for a floating two terminal admittance component are added. This routine also works if component is grounded. Positive elements are placed at [Node1,Node2] and [Node2,Node1]. This routine is only to be used after **spCreate**() (p. 49) and before **spMNA_Preorder**() (p. 57), **spFactor**() (p. 51) or **spOrderAndFactor**() (p. 59).

Returns :

Error code. Possible errors include *spNO_MEMORY*. Error is not cleared in this routine.

Parameters:

Matrix Pointer to the matrix that component is to be entered in.

Node1 Row and column indices for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Node zero is the ground node. In no case may *Node1* be less than zero.

Node2 Row and column indices for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Node zero is the ground node. In no case may *Node2* be less than zero.

Template Collection of pointers to four elements that are later used to directly address elements. User must supply the template, this routine will fill it.

4.6.4.17 spcEXTERN spElement* spGetElement (spMatrix *eMatrix*, int *Row*, int *Col*)

Finds element [Row,Col] and returns a pointer to it. If element is not found then it is created and spliced into matrix. This routine is only to be used after **spCreate**() (p. 49) and before **spMNA_Preorder**() (p. 57), **spFactor**() (p. 51) or **spOrderAndFactor**() (p. 59). Returns a pointer to the real portion of an **spElement**. This pointer is later used by *spADD_xxx_ELEMENT* to directly access element.

Returns :

Returns a pointer to the element. This pointer is then used to directly access the element during successive builds.

Parameters:

eMatrix Pointer to the matrix that the element is to be added to.

Row Row index for element. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Elements placed in row zero are discarded. In no case may *Row* be less than zero.

Col Column index for element. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Elements placed in column zero are discarded. In no case may *Col* be less than zero.

See also:

spFindElement() (p. 53)

4.6.4.18 spcEXTERN spGenericPtr spGetInitInfo (spElement * *pElement*)

This function returns a pointer to a data structure that is used to contain initialization information to a matrix element.

Returns :

The pointer to the initialiation information data structure that is associated with a particular matrix element.

Parameters:

pElement Pointer to the matrix element.

See also:

spInitialize()

4.6.4.19 spcEXTERN spError spGetOnes (spMatrix *Matrix*, int *Pos*, int *Neg*, int *Eqn*, struct spTemplate * *Template*)

Addition of four structural ones to matrix by index. Performs similar function to **spGetQuad**() (p. 55) except this routine is meant for components that do not have an admittance representation.

The following stamp is used:

```
       Pos  Neg  Eqn
Pos  [  .    .    1  ]
Neg  [  .    .   -1  ]
Eqn  [  1   -1    .  ]
```

Returns :
Error code. Possible errors include *spNO_MEMORY*. Error is not cleared in this routine.

Parameters:

Matrix Pointer to the matrix that component is to be entered in.

Pos See stamp above. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may *Pos* be less than zero.

Neg See stamp above. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may *Neg* be less than zero.

Eqn See stamp above. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may *Eqn* be less than zero.

Template Collection of pointers to four elements that are later used to directly address elements. User must supply the template, this routine will fill it.

4.6.4.20 spcEXTERN spError spGetQuad (spMatrix *Matrix*, int *Row1*, int *Row2*, int *Col1*, int *Col2*, struct spTemplate * *Template*)

Similar to **spGetAdmittance**() (p. 53), except that **spGetAdmittance**() (p. 53) only handles 2-terminal components, whereas **spGetQuad**() (p. 55) handles simple 4-terminals as well. These 4-terminals are simply generalized 2-terminals with the option of having the sense terminals different from the source and sink terminals. **spGetQuad**() (p. 55) adds four elements to the matrix. Positive elements occur at [Row1,Col1] [Row2,Col2] while negative elements occur at [Row1,Col2] and [Row2,Col1]. The routine works fine if any of the rows and columns are zero. This routine is only to be used after **spCreate**() (p. 49) and before **spMNA_Preorder**() (p. 57), **spFactor**() (p. 51) or **spOrderAndFactor**() (p. 59) unless *TRANSLATE* is set true.

Returns :
Error code. Possible errors include *spNO_MEMORY*. Error is not cleared in this routine.

Parameters:

Matrix Pointer to the matrix that component is to be entered in.

Row1 First row index for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may Row1 be less than zero.

Row2 Second row index for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground row. In no case may Row2 be less than zero.

Col1 First column index for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground column. In no case may Col1 be less than zero.

Col2 Second column index for elements. Must be in the range of [0..Size] unless the options *EXPANDABLE* or *TRANSLATE* are used. Zero is the ground column. In no case may Col2 be less than zero.

Template Collection of pointers to four elements that are later used to directly address elements. User must supply the template, this routine will fill it.

4.6.4.21 spcEXTERN int spGetSize (spMatrix *eMatrix*, int *External*)

Returns the size of the matrix. Either the internal or external size of the matrix is returned.

Parameters:

eMatrix Pointer to matrix.

External If *External* is set true, the external size , i.e., the value of the largest external row or column number encountered is returned. Otherwise the true size of the matrix is returned. These two sizes may differ if the *TRANSLATE* option is set true.

4.6.4.22 spcEXTERN void spInstallInitInfo (spElement * *pElement*, spGenericPtr *pInitInfo*)

This function installs a pointer to a data structure that is used to contain initialization information to a matrix element. It is is then used by **spInitialize**() to initialize the matrix.

Parameters:

pElement Pointer to matrix element.

pInitInfo Pointer to the data structure that will contain initialiation information.

See also:

spInitialize()

4.6.4.23 spcEXTERN spREAL spLargestElement (spMatrix *eMatrix*)

This routine, along with **spRoundoff**() (p. 61), are used to gauge the stability of a factorization. If the factorization is determined to be too unstable, then the matrix should be reordered. The routines compute quantities that are needed in the computation of a bound on the error attributed to any one element in the matrix during the factorization. In other words, there is a matrix $E = [e_{ij}]$ of error terms such that $A + E = LU$. This routine finds a bound on $|e_{ij}|$. Erisman & Reid [1] showed that $|e_{ij}| < 3.01u\rho m_{ij}$, where u is the machine rounding unit, $\rho = \max a_{ij}$ where the max is taken over every row i, column j, and step k, and m_{ij} is the number of multiplications required in the computation of l_{ij} if $i > j$ or u_{ij} otherwise. Barlow [2] showed that $\rho < \max_i ||l_i||_p \max_j ||u_j||_q$ where $1/p + 1/q = 1$.

spLargestElement() (p. 56) finds the magnitude on the largest element in the matrix. If the matrix has not yet been factored, the largest element is found by direct search. If the matrix is factored, a bound on the largest element in any of the reduced submatrices is computed using Barlow with $p = \infty$ and $q = 1$. The ratio of these two numbers is the growth, which can be used to determine if the pivoting order is adequate. A large growth implies that considerable error has been made in the factorization and that it is probably a good idea to reorder the matrix. If a large growth in encountered after using **spFactor**() (p. 51), reconstruct the matrix and refactor using **spOrderAndFactor**() (p. 59). If a large growth is encountered after using **spOrderAndFactor**() (p. 59), refactor using **spOrderAndFactor**() (p. 59) with the pivot threshold increased, say to 0.1.

Using only the size of the matrix as an upper bound on m_{ij} and Barlow's bound, the user can estimate the size of the matrix error terms e_{ij} using the bound of Erisman and Reid. **spRoundoff**() (p. 61) computes a tighter bound (with more work) based on work by Gear [3], $|e_{ij}| < 1.01u\rho(tc^3 + (1+t)c^2)$ where t is the threshold and c is the maximum number of off-diagonal elements in any row of L. The expensive part of computing this bound is determining the maximum number of off-diagonals in L, which changes only when the order of the matrix changes. This number is computed and saved, and only recomputed if the matrix is reordered.

[1] A. M. Erisman, J. K. Reid. Monitoring the stability of the triangular factorization of a sparse matrix. Numerische Mathematik. Vol. 22, No. 3, 1974, pp 183-186.

[2] J. L. Barlow. A note on monitoring the stability of triangular decomposition of sparse matrices. "SIAM Journal of Scientific and Statistical Computing." Vol. 7, No. 1, January 1986, pp 166-168.

[3] I. S. Duff, A. M. Erisman, J. K. Reid. "Direct Methods for Sparse Matrices." Oxford 1986. pp 99.

Returns :

If matrix is not factored, returns the magnitude of the largest element in the matrix. If the matrix is factored, a bound on the magnitude of the largest element in any of the reduced submatrices is returned.

Parameters:

eMatrix Pointer to the matrix.

4.6.4.24 spcEXTERN void spMNA_Preorder (spMatrix *eMatrix*)

This routine massages modified node admittance matrices to remove zeros from the diagonal. It takes advantage of the fact that the row and column associated with a zero diagonal usually have structural ones placed symmetricly. This routine should be used only on modified node admittance matrices and should be executed after the matrix has been built but before the factorization begins. It should be executed for the initial factorization only and should be executed before the rows have been linked. Thus it should be run before using **spScale**() (p. 62), **spMultiply**() (p. 58), **spDeleteRowAndCol**() (p. 50), or **spNorm**() (p. 59).

This routine exploits the fact that the structural ones are placed in the matrix in symmetric twins. For example, the stamps for grounded and a floating voltage sources are

```
grounded:              floating:
[  x    x    1 ]       [  x    x    1 ]
[  x    x      ]       [  x    x   -1 ]
[  1           ]       [  1   -1      ]
```

Notice for the grounded source, there is one set of twins, and for the floating, there are two sets. We remove the zero from the diagonal by swapping the rows associated with a set of twins. For example:

```
grounded:              floating 1:            floating 2:
[  1           ]       [  1   -1      ]       [  x    x    1 ]
[  x    x      ]       [  x    x   -1 ]       [  1   -1      ]
[  x    x    1 ]       [  x    x    1 ]       [  x    x   -1 ]
```

It is important to deal with any zero diagonals that only have one set of twins before dealing with those that have more than one because swapping row destroys the symmetry of any twins in the rows being swapped, which may limit future moves. Consider

```
[  x    x    1      ]
[  x    x   -1    1 ]
[  1   -1           ]
[       1           ]
```

There is one set of twins for diagonal 4 and two for diagonal 3. Dealing with diagonal 4 first requires swapping rows 2 and 4.

```
[  x   x   1     ]
[      1         ]
[  1  -1         ]
[  x   x  -1   1 ]
```

We can now deal with diagonal 3 by swapping rows 1 and 3.

```
[  1  -1         ]
[      1         ]
[  x   x   1     ]
[  x   x  -1   1 ]
```

And we are done, there are no zeros left on the diagonal. However, if we originally dealt with diagonal 3 first, we could swap rows 2 and 3

```
[  x   x   1     ]
[  1  -1         ]
[  x   x  -1   1 ]
[      1         ]
```

Diagonal 4 no longer has a symmetric twin and we cannot continue.

So we always take care of lone twins first. When none remain, we choose arbitrarily a set of twins for a diagonal with more than one set and swap the rows corresponding to that twin. We then deal with any lone twins that were created and repeat the procedure until no zero diagonals with symmetric twins remain.

In this particular implementation, columns are swapped rather than rows. The algorithm used in this function was developed by Ken Kundert and Tom Quarles.

Parameters:

eMatrix Pointer to the matrix to be preordered.

4.6.4.25 spcEXTERN void spMultiply (spMatrix *eMatrix*, spREAL *RHS*[], spREAL *Solution*[])

Multiplies matrix by solution vector to find source vector. Assumes matrix has not been factored. This routine can be used as a test to see if solutions are correct. It should not be used before **spMNA_Preorder()** (p. 57).

Parameters:

eMatrix Pointer to the matrix.

RHS RHS is the right hand side. This is what is being solved for.

Solution Solution is the vector being multiplied by the matrix.

iRHS iRHS is the imaginary portion of the right hand side. This is what is being solved for. This is only necessary if the matrix is complex and *spSEPARATED_COMPLEX_VECTORS* is true.

iSolution iSolution is the imaginary portion of the vector being multiplied by the matrix. This is only necessary if the matrix is complex and *spSEPARATED_COMPLEX_VECTORS* is true.

4.6.4.26 spcEXTERN void spMultTransposed (spMatrix *eMatrix*, spREAL *RHS*[], spREAL *Solution*[])

Multiplies transposed matrix by solution vector to find source vector. Assumes matrix has not been factored. This routine can be used as a test to see if solutions are correct. It should not be used before **spMNA_Preorder()** (p. 57).

Parameters:

eMatrix Pointer to the matrix.

RHS RHS is the right hand side. This is what is being solved for.

Solution Solution is the vector being multiplied by the matrix.

iRHS iRHS is the imaginary portion of the right hand side. This is what is being solved for. This is only necessary if the matrix is complex and *spSEPARATED_COMPLEX_VECTORS* is true.

iSolution iSolution is the imaginary portion of the vector being multiplied by the matrix. This is only necessary if the matrix is complex and *spSEPARATED_COMPLEX_VECTORS* is true.

4.6.4.27 spcEXTERN spREAL spNorm (spMatrix *eMatrix*)

Computes the L-infinity norm of an unfactored matrix. It is a fatal error to pass this routine a factored matrix.

Returns :

The largest absolute row sum of matrix.

Parameters:

eMatrix Pointer to the matrix.

4.6.4.28 spcEXTERN spError spOrderAndFactor (spMatrix *eMatrix*, spREAL *RHS*[], spREAL *RelThreshold*, spREAL *AbsThreshold*, int *DiagPivoting*)

This routine chooses a pivot order for the matrix and factors it into *LU* form. It handles both the initial factorization and subsequent factorizations when a reordering is desired. This is handled in a manner that is transparent to the user. The routine uses a variation of Gauss's method where the pivots are associated with *L* and the diagonal terms of *U* are one.

Returns :

The error code is returned. Possible errors are *spNO_MEMORY*, *spSINGULAR* and *spSMALL_PIVOT*. Error is cleared upon entering this function.

Parameters:

eMatrix Pointer to the matrix.

RHS Representative right-hand side vector that is used to determine pivoting order when the right hand side vector is sparse. If RHS is a NULL pointer then the RHS vector is assumed to be full and it is not used when determining the pivoting order.

RelThreshold This number determines what the pivot relative threshold will be. It should be between zero and one. If it is one then the pivoting method becomes complete pivoting, which is very slow and tends to fill up the matrix. If it is set close to zero the pivoting method becomes strict Markowitz with no threshold. The pivot threshold is used to eliminate pivot

candidates that would cause excessive element growth if they were used. Element growth is the cause of roundoff error. Element growth occurs even in well-conditioned matrices. Setting the *RelThreshold* large will reduce element growth and roundoff error, but setting it too large will cause execution time to be excessive and will result in a large number of fill-ins. If this occurs, accuracy can actually be degraded because of the large number of operations required on the matrix due to the large number of fill-ins. A good value seems to be 0.001. The default is chosen by giving a value larger than one or less than or equal to zero. This value should be increased and the matrix resolved if growth is found to be excessive. Changing the pivot threshold does not improve performance on matrices where growth is low, as is often the case with ill-conditioned matrices. Once a valid threshold is given, it becomes the new default. The default value of *RelThreshold* was choosen for use with nearly diagonally dominant matrices such as node- and modified-node admittance matrices. For these matrices it is usually best to use diagonal pivoting. For matrices without a strong diagonal, it is usually best to use a larger threshold, such as 0.01 or 0.1.

AbsThreshold The absolute magnitude an element must have to be considered as a pivot candidate, except as a last resort. This number should be set significantly smaller than the smallest diagonal element that is is expected to be placed in the matrix. If there is no reasonable prediction for the lower bound on these elements, then *AbsThreshold* should be set to zero. *AbsThreshold* is used to reduce the possibility of choosing as a pivot an element that has suffered heavy cancellation and as a result mainly consists of roundoff error. Once a valid threshold is given, it becomes the new default.

DiagPivoting A flag indicating that pivot selection should be confined to the diagonal if possible. If *DiagPivoting* is nonzero and if *DIAGONAL_PIVOTING* is enabled pivots will be chosen only from the diagonal unless there are no diagonal elements that satisfy the threshold criteria. Otherwise, the entire reduced submatrix is searched when looking for a pivot. The diagonal pivoting in Sparse is efficient and well refined, while the off-diagonal pivoting is not. For symmetric and near symmetric matrices, it is best to use diagonal pivoting because it results in the best performance when reordering the matrix and when factoring the matrix without ordering. If there is a considerable amount of nonsymmetry in the matrix, then off-diagonal pivoting may result in a better equation ordering simply because there are more pivot candidates to choose from. A better ordering results in faster subsequent factorizations. However, the initial pivot selection process takes considerably longer for off-diagonal pivoting.

See also:

spFactor() (p. 51)

4.6.4.29 spcEXTERN void spPartition (spMatrix *eMatrix*, int *Mode*)

This routine determines the cost to factor each row using both direct and indirect addressing and decides, on a row-by-row basis, which addressing mode is fastest. This information is used in **spFactor**() (p. 51) to speed the factorization.

When factoring a previously ordered matrix using **spFactor**() (p. 51), Sparse operates on a row-at-a-time basis. For speed, on each step, the row being updated is copied into a full vector and the operations are performed on that vector. This can be done one of two ways, either using direct addressing or indirect addressing. Direct addressing is fastest when the matrix is relatively dense and indirect addressing is best when the matrix is quite sparse. The user selects the type of partition used with *Mode*. If *Mode* is set to *spDIRECT_PARTITION*, then the all rows are placed in the direct addressing partition. Similarly, if *Mode* is set to *spINDIRECT_PARTITION*, then the all rows are placed in the indirect addressing partition. By setting *Mode* to *spAUTO_PARTITION*, the user allows Sparse to select the partition for each row individually. **spFactor**() (p. 51) generally runs faster if Sparse is allowed to choose its own partitioning, however choosing a partition is expensive. The time required

to choose a partition is of the same order of the cost to factor the matrix. If you plan to factor a large number of matrices with the same structure, it is best to let Sparse choose the partition. Otherwise, you should choose the partition based on the predicted density of the matrix.

Parameters:

eMatrix Pointer to matrix.

Mode Mode must be one of three special codes: *spDIRECT_PARTITION*, *spINDIRECT_PARTITION*, or *spAUTO_PARTITION*.

4.6.4.30 spcEXTERN void spPrint (spMatrix *eMatrix*, int *PrintReordered*, int *Data*, int *Header*)

Formats and send the matrix to standard output. Some elementary statistics are also output. The matrix is output in a format that is readable by people.

Parameters:

eMatrix Pointer to matrix.

PrintReordered Indicates whether the matrix should be printed out in its original form, as input by the user, or whether it should be printed in its reordered form, as used by the matrix routines. A zero indicates that the matrix should be printed as inputed, a one indicates that it should be printed reordered.

Data Boolean flag that when false indicates that output should be compressed such that only the existence of an element should be indicated rather than giving the actual value. Thus 11 times as many can be printed on a row. A zero signifies that the matrix should be printed compressed. A one indicates that the matrix should be printed in all its glory.

Header Flag indicating that extra information should be given, such as row and column numbers.

4.6.4.31 spcEXTERN spREAL spPseudoCondition (spMatrix *eMatrix*)

Computes the magnitude of the ratio of the largest to the smallest pivots. This quantity is an indicator of ill-conditioning in the matrix. If this ratio is large, and if the matrix is scaled such that uncertainties in the RHS and the matrix entries are equilibrated, then the matrix is ill-conditioned. However, a small ratio does not necessarily imply that the matrix is well-conditioned. This routine must only be used after a matrix has been factored by **spOrderAndFactor**() (p. 59) or **spFactor**() (p. 51) and before it is cleared by **spClear**() (p. 49) or **spInitialize**(). The pseudocondition is faster to compute than the condition number calculated by **spCondition**() (p. 49), but is not as informative.

Returns :

The magnitude of the ratio of the largest to smallest pivot used during previous factorization. If the matrix was singular, zero is returned.

Parameters:

eMatrix Pointer to the matrix.

4.6.4.32 spcEXTERN spREAL spRoundoff (spMatrix *eMatrix*, spREAL *Rho*)

This routine, along with **spLargestElement**() (p. 56), are used to gauge the stability of a factorization. See description of **spLargestElement**() (p. 56) for more information.

Returns :

Returns a bound on the magnitude of the largest element in $E = A - LU$.

Parameters:

eMatrix Pointer to the matrix.

Rho The bound on the magnitude of the largest element in any of the reduced submatrices. This is the number computed by the function **spLargestElement**() (p. 56) when given a factored matrix. If this number is negative, the bound will be computed automatically.

4.6.4.33 spcEXTERN void spScale (spMatrix *eMatrix*, spREAL *RHS_ScaleFactors*[], spREAL *SolutionScaleFactors*[])

This function scales the matrix to enhance the possibility of finding a good pivoting order. Note that scaling enhances accuracy of the solution only if it affects the pivoting order, so it makes no sense to scale the matrix before **spFactor**() (p. 51). If scaling is desired it should be done before **spOrderAndFactor**() (p. 59). There are several things to take into account when choosing the scale factors. First, the scale factors are directly multiplied against the elements in the matrix. To prevent roundoff, each scale factor should be equal to an integer power of the number base of the machine. Since most machines operate in base two, scale factors should be a power of two. Second, the matrix should be scaled such that the matrix of element uncertainties is equilibrated. Third, this function multiplies the scale factors by the elements, so if one row tends to have uncertainties 1000 times smaller than the other rows, then its scale factor should be 1024, not 1/1024. Fourth, to save time, this function does not scale rows or columns if their scale factors are equal to one. Thus, the scale factors should be normalized to the most common scale factor. Rows and columns should be normalized separately. For example, if the size of the matrix is 100 and 10 rows tend to have uncertainties near 1e-6 and the remaining 90 have uncertainties near 1e-12, then the scale factor for the 10 should be 1/1,048,576 and the scale factors for the remaining 90 should be 1. Fifth, since this routine directly operates on the matrix, it is necessary to apply the scale factors to the RHS and Solution vectors. It may be easier to simply use **spOrderAndFactor**() (p. 59) on a scaled matrix to choose the pivoting order, and then throw away the matrix. Subsequent factorizations, performed with **spFactor**() (p. 51), will not need to have the RHS and Solution vectors descaled. Lastly, this function should not be executed before the function **spMNA_Preorder**() (p. 57).

Parameters:

eMatrix Pointer to the matrix to be scaled.

SolutionScaleFactors The array of Solution scale factors. These factors scale the columns. All scale factors are real valued.

RHS_ScaleFactors The array of RHS scale factors. These factors scale the rows. All scale factors are real valued.

4.6.4.34 spcEXTERN void spSetComplex (spMatrix *eMatrix*)

Forces matrix to be complex.

Parameters:

eMatrix Pointer to matrix.

4.6.4.35 spcEXTERN void spSetReal (spMatrix *eMatrix*)

Forces matrix to be real.

Parameters:

eMatrix Pointer to matrix.

4.6.4.36 spcEXTERN void spSolve (spMatrix *eMatrix*, spREAL *RHS*[ ], spREAL *Solution*[ ])

Performs forward elimination and back substitution to find the unknown vector from the *RHS* vector and factored matrix. This routine assumes that the pivots are associated with the lower triangular matrix and that the diagonal of the upper triangular matrix consists of ones. This routine arranges the computation in different way than is traditionally used in order to exploit the sparsity of the right-hand side. See the reference in spRevision.

Parameters:

eMatrix Pointer to matrix.

RHS *RHS* is the input data array, the right hand side. This data is undisturbed and may be reused for other solves.

Solution *Solution* is the output data array. This routine is constructed such that *RHS* and *Solution* can be the same array.

iRHS *iRHS* is the imaginary portion of the input data array, the right hand side. This data is undisturbed and may be reused for other solves. This argument is only necessary if matrix is complex and if *spSEPARATED_COMPLEX_VECTOR* is set true.

iSolution *iSolution* is the imaginary portion of the output data array. This routine is constructed such that *iRHS* and *iSolution* can be the same array. This argument is only necessary if matrix is complex and if *spSEPARATED_COMPLEX_VECTOR* is set true.

4.6.4.37 spcEXTERN void spSolveTransposed (spMatrix *eMatrix*, spREAL *RHS*[ ], spREAL *Solution*[ ])

Performs forward elimination and back substitution to find the unknown vector from the RHS vector and transposed factored matrix. This routine is useful when performing sensitivity analysis on a circuit using the adjoint method. This routine assumes that the pivots are associated with the untransposed lower triangular matrix and that the diagonal of the untransposed upper triangular matrix consists of ones.

Parameters:

eMatrix Pointer to matrix.

RHS *RHS* is the input data array, the right hand side. This data is undisturbed and may be reused for other solves.

Solution *Solution* is the output data array. This routine is constructed such that *RHS* and *Solution* can be the same array.

iRHS *iRHS* is the imaginary portion of the input data array, the right hand side. This data is undisturbed and may be reused for other solves. If *spSEPARATED_COMPLEX_VECTOR* is set false, or if matrix is real, there is no need to supply this array.

iSolution *iSolution* is the imaginary portion of the output data array. This routine is constructed such that *iRHS* and *iSolution* can be the same array. If *spSEPARATED_COMPLEX_VECTOR* is set false, or if matrix is real, there is no need to supply this array.

4.6.4.38 spcEXTERN void spStripFills (spMatrix *eMatrix*)

Strips the matrix of all fill-ins.

Parameters:

eMatrix Pointer to the matrix to be stripped.

4.6.4.39 spcEXTERN void spWhereSingular (spMatrix *eMatrix*, int * *pRow*, int * *pCol*)

This function returns the row and column number where the matrix was detected as singular (if pivoting was allowed on the last factorization) or where a zero was detected on the diagonal (if pivoting was not allowed on the last factorization). Pivoting is performed only in **spOrderAndFactor**() (p. 59).

Parameters:

eMatrix The matrix for which the error status is desired.

pRow The row number.

pCol The column number.

4.7 spOutput.c File Reference

```
#include <stdio.h>
#include "spConfig.h"
#include "spMatrix.h"
#include "spDefs.h"
```

Functions

- void **spPrint** (**spMatrix** eMatrix, int PrintReordered, int Data, int Header)
- int **spFileMatrix** (**spMatrix** eMatrix, char *File, char *Label, int Reordered, int Data, int Header)
- int **spFileVector** (**spMatrix** eMatrix, char *File, spREAL RHS[])
- int **spFileStats** (**spMatrix** eMatrix, char *File, char *Label)

4.7.1 Detailed Description

This file contains the output-to-file and output-to-screen routines for the matrix package.

Objects that begin with the *spc* prefix are considered private and should not be used.

Author:
Kenneth S. Kundert <kundert@users.sourceforge.net>

4.7.2 Function Documentation

4.7.2.1 int spFileMatrix (spMatrix *eMatrix*, char * *File*, char * *Label*, int *Reordered*, int *Data*, int *Header*)

Writes matrix to file in format suitable to be read back in by the matrix test program.

Returns :
One is returned if routine was successful, otherwise zero is returned. The calling function can query *errno* (the system global error variable) as to the reason why this routine failed.

Parameters:
eMatrix Pointer to matrix.

File Name of file into which matrix is to be written.

Label String that is transferred to file and is used as a label.

Reordered Specifies whether matrix should be output in reordered form, or in original order.

Data Indicates that the element values should be output along with the indices for each element. This parameter must be true if matrix is to be read by the sparse test program.

Header Indicates that header is desired. This parameter must be true if matrix is to be read by the sparse test program.

4.7.2.2 int spFileStats (spMatrix *eMatrix*, char * *File*, char * *Label*)

Writes useful information concerning the matrix to a file. Should be executed after the matrix is factored.

Returns :

One is returned if routine was successful, otherwise zero is returned. The calling function can query *errno* (the system global error variable) as to the reason why this routine failed.

Parameters:

eMatrix Pointer to matrix.

File Name of file into which matrix is to be written.

Label String that is transferred to file and is used as a label.

4.7.2.3 int spFileVector (spMatrix *eMatrix*, char * *File*, spREAL *RHS*[])

Writes vector to file in format suitable to be read back in by the matrix test program. This routine should be executed after the function spFileMatrix.

Returns :

One is returned if routine was successful, otherwise zero is returned. The calling function can query *errno* (the system global error variable) as to the reason why this routine failed.

Parameters:

eMatrix Pointer to matrix.

File Name of file into which matrix is to be written.

RHS Right-hand side vector. This is only the real portion if *spSEPARATED_COMPLEX_VECTORS* is true.

iRHS Right-hand side vector, imaginary portion. Not necessary if matrix is real or if *spSEPARATED_COMPLEX_VECTORS* is set false. *iRHS* is a macro that replaces itself with ', iRHS' if the options *spCOMPLEX* and *spSEPARATED_COMPLEX_VECTORS* are set, otherwise it disappears without a trace.

4.7.2.4 void spPrint (spMatrix *eMatrix*, int *PrintReordered*, int *Data*, int *Header*)

Formats and send the matrix to standard output. Some elementary statistics are also output. The matrix is output in a format that is readable by people.

Parameters:

eMatrix Pointer to matrix.

PrintReordered Indicates whether the matrix should be printed out in its original form, as input by the user, or whether it should be printed in its reordered form, as used by the matrix routines. A zero indicates that the matrix should be printed as inputed, a one indicates that it should be printed reordered.

Data Boolean flag that when false indicates that output should be compressed such that only the existence of an element should be indicated rather than giving the actual value. Thus 11 times as many can be printed on a row. A zero signifies that the matrix should be printed compressed. A one indicates that the matrix should be printed in all its glory.

Header Flag indicating that extra information should be given, such as row and column numbers.

4.8 spSolve.c File Reference

```
#include <stdio.h>
#include "spConfig.h"
#include "spMatrix.h"
#include "spDefs.h"
```

Functions

- void **spSolve** (**spMatrix** eMatrix, spREAL RHS[], spREAL Solution[])
- void **spSolveTransposed** (**spMatrix** eMatrix, spREAL RHS[], spREAL Solution[])

4.8.1 Detailed Description

This file contains the forward and backward substitution routines for the sparse matrix routines.

Objects that begin with the *spc* prefix are considered private and should not be used.

Author:
Kenneth S. Kundert <kundert@users.sourceforge.net>

4.8.2 Function Documentation

4.8.2.1 void spSolve (spMatrix *eMatrix*, spREAL *RHS*[], spREAL *Solution*[])

Performs forward elimination and back substitution to find the unknown vector from the *RHS* vector and factored matrix. This routine assumes that the pivots are associated with the lower triangular matrix and that the diagonal of the upper triangular matrix consists of ones. This routine arranges the computation in different way than is traditionally used in order to exploit the sparsity of the right-hand side. See the reference in spRevision.

Parameters:

eMatrix Pointer to matrix.

RHS *RHS* is the input data array, the right hand side. This data is undisturbed and may be reused for other solves.

Solution *Solution* is the output data array. This routine is constructed such that *RHS* and *Solution* can be the same array.

iRHS *iRHS* is the imaginary portion of the input data array, the right hand side. This data is undisturbed and may be reused for other solves. This argument is only necessary if matrix is complex and if *spSEPARATED_COMPLEX_VECTOR* is set true.

iSolution *iSolution* is the imaginary portion of the output data array. This routine is constructed such that *iRHS* and *iSolution* can be the same array. This argument is only necessary if matrix is complex and if *spSEPARATED_COMPLEX_VECTOR* is set true.

4.8.2.2 void spSolveTransposed (spMatrix *eMatrix*, spREAL *RHS*[], spREAL *Solution*[])

Performs forward elimination and back substitution to find the unknown vector from the RHS vector and transposed factored matrix. This routine is useful when performing sensitivity analysis on a circuit

using the adjoint method. This routine assumes that the pivots are associated with the untransposed lower triangular matrix and that the diagonal of the untransposed upper triangular matrix consists of ones.

Parameters:

eMatrix Pointer to matrix.

RHS *RHS* is the input data array, the right hand side. This data is undisturbed and may be reused for other solves.

Solution *Solution* is the output data array. This routine is constructed such that *RHS* and *Solution* can be the same array.

iRHS *iRHS* is the imaginary portion of the input data array, the right hand side. This data is undisturbed and may be reused for other solves. If *spSEPARATED_COMPLEX_VECTOR* is set false, or if matrix is real, there is no need to supply this array.

iSolution *iSolution* is the imaginary portion of the output data array. This routine is constructed such that *iRHS* and *iSolution* can be the same array. If *spSEPARATED_COMPLEX_VECTOR* is set false, or if matrix is real, there is no need to supply this array.

4.9 spUtils.c File Reference

```
#include <stdio.h>
#include "spConfig.h"
#include "spMatrix.h"
#include "spDefs.h"
```

Defines

- #define **spINSIDE_SPARSE**
- #define **NORM**(a) (nr = ABS((a).Real), ni = ABS((a).Imag), MAX (nr,ni))
- #define **SLACK** 1e4

Functions

- void **spMNA_Preorder** (**spMatrix** eMatrix)
- void **spScale** (**spMatrix** eMatrix, spREAL RHS_ScaleFactors[], spREAL SolutionScaleFactors[])
- void **spMultiply** (**spMatrix** eMatrix, spREAL RHS[], spREAL Solution[])
- void **spMultTransposed** (**spMatrix** eMatrix, spREAL RHS[], spREAL Solution[])
- void **spDeterminant** (**spMatrix** eMatrix, int *pExponent, spREAL *pDeterminant, spREAL *piDeterminant)
- void **spStripFills** (**spMatrix** eMatrix)
- void **spDeleteRowAndCol** (**spMatrix** eMatrix, int Row, int Col)
- spREAL **spPseudoCondition** (**spMatrix** eMatrix)
- spREAL **spCondition** (**spMatrix** eMatrix, spREAL NormOfMatrix, int *pError)
- spREAL **spNorm** (**spMatrix** eMatrix)
- spREAL **spLargestElement** (**spMatrix** eMatrix)
- spREAL **spRoundoff** (**spMatrix** eMatrix, spREAL Rho)
- void **spErrorMessage** (**spMatrix** eMatrix, FILE *Stream, char *Originator)

4.9.1 Detailed Description

This file contains various optional utility routines.

Objects that begin with the *spc* prefix are considered private and should not be used.

Author:
Kenneth S. Kundert <kundert@users.sourceforge.net>

4.9.2 Function Documentation

4.9.2.1 spREAL spCondition (spMatrix *eMatrix*, spREAL *NormOfMatrix*, int * *pError*)

Computes an estimate of the condition number using a variation on the LINPACK condition number estimation algorithm. This quantity is an indicator of ill-conditioning in the matrix. To avoid problems with overflow, the reciprocal of the condition number is returned. If this number is small, and if the

matrix is scaled such that uncertainties in the RHS and the matrix entries are equilibrated, then the matrix is ill-conditioned. If the this number is near one, the matrix is well conditioned. This routine must only be used after a matrix has been factored by **spOrderAndFactor**() (p. 24) or **spFactor**() (p. 23) and before it is cleared by **spClear**() (p. 11) or **spInitialize**() (p. 14).

Unlike the LINPACK condition number estimator, this routines returns the L infinity condition number. This is an artifact of Sparse placing ones on the diagonal of the upper triangular matrix rather than the lower. This difference should be of no importance.

References:

A.K. Cline, C.B. Moler, G.W. Stewart, J.H. Wilkinson. An estimate for the condition number of a matrix. SIAM Journal on Numerical Analysis. Vol. 16, No. 2, pages 368-375, April 1979.

J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart. LINPACK User's Guide. SIAM, 1979.

Roger G. Grimes, John G. Lewis. Condition number estimation for sparse matrices. SIAM Journal on Scientific and Statistical Computing. Vol. 2, No. 4, pages 384-388, December 1981.

Dianne Prost O'Leary. Estimating matrix condition numbers. SIAM Journal on Scientific and Statistical Computing. Vol. 1, No. 2, pages 205-209, June 1980.

Returns :

The reciprocal of the condition number. If the matrix was singular, zero is returned.

Parameters:

eMatrix Pointer to the matrix.

NormOfMatrix The L-infinity norm of the unfactored matrix as computed by **spNorm**() (p. 74).

pError Used to return error code. Possible errors include *spSINGULAR* or *spNO_MEMORY*.

4.9.2.2 void spDeleteRowAndCol (spMatrix *eMatrix*, int *Row*, int *Col*)

Deletes a row and a column from a matrix.

Sparse will abort if an attempt is made to delete a row or column that doesn't exist.

Parameters:

eMatrix Pointer to the matrix in which the row and column are to be deleted.

Row Row to be deleted.

Col Column to be deleted.

4.9.2.3 void spDeterminant (spMatrix *eMatrix*, int * *pExponent*, spREAL * *pDeterminant*, spREAL * *piDeterminant*)

This routine in capable of calculating the determinant of the matrix once the LU factorization has been performed. Hence, only use this routine after **spFactor**() (p. 23) and before **spClear**() (p. 11). The determinant equals the product of all the diagonal elements of the lower triangular matrix L, except that this product may need negating. Whether the product or the negative product equals the determinant is determined by the number of row and column interchanges performed. Note that the determinants of matrices can be very large or very small. On large matrices, the determinant can be far larger or smaller than can be represented by a floating point number. For this reason the determinant is scaled to a reasonable value and the logarithm of the scale factor is returned.

Parameters:

eMatrix A pointer to the matrix for which the determinant is desired.

pExponent The logarithm base 10 of the scale factor for the determinant. To find the actual determinant, Exponent should be added to the exponent of Determinant.

pDeterminant The real portion of the determinant. This number is scaled to be greater than or equal to 1.0 and less than 10.0.

piDeterminant The imaginary portion of the determinant. When the matrix is real this pointer need not be supplied, nothing will be returned. This number is scaled to be greater than or equal to 1.0 and less than 10.0.

4.9.2.4 void spErrorMessage (spMatrix *eMatrix*, FILE * *Stream*, char * *Originator*)

This routine prints a short message describing the error error state of sparse. No message is produced if there is no error. The error state is cleared.

Parameters:

eMatrix Matrix for which the error message is to be printed.

Stream Stream to which the error message is to be printed.

Originator Name of originator of error message. If NULL, 'sparse' is used. If zero-length string, no originator is printed.

4.9.2.5 spREAL spLargestElement (spMatrix *eMatrix*)

This routine, along with **spRoundoff**() (p. 75), are used to gauge the stability of a factorization. If the factorization is determined to be too unstable, then the matrix should be reordered. The routines compute quantities that are needed in the computation of a bound on the error attributed to any one element in the matrix during the factorization. In other words, there is a matrix $E = [e_{ij}]$ of error terms such that $A + E = LU$. This routine finds a bound on $|e_{ij}|$. Erisman & Reid [1] showed that $|e_{ij}| < 3.01u\rho m_{ij}$, where u is the machine rounding unit, $\rho = \max a_{ij}$ where the max is taken over every row i, column j, and step k, and m_{ij} is the number of multiplications required in the computation of l_{ij} if $i > j$ or u_{ij} otherwise. Barlow [2] showed that $\rho < \max_i ||l_i||_p \max_j ||u_j||_q$ where $1/p + 1/q = 1$.

spLargestElement() (p. 71) finds the magnitude on the largest element in the matrix. If the matrix has not yet been factored, the largest element is found by direct search. If the matrix is factored, a bound on the largest element in any of the reduced submatrices is computed using Barlow with $p = \infty$ and $q = 1$. The ratio of these two numbers is the growth, which can be used to determine if the pivoting order is adequate. A large growth implies that considerable error has been made in the factorization and that it is probably a good idea to reorder the matrix. If a large growth in encountered after using **spFactor**() (p. 23), reconstruct the matrix and refactor using **spOrderAndFactor**() (p. 24). If a large growth is encountered after using **spOrderAndFactor**() (p. 24), refactor using **spOrderAndFactor**() (p. 24) with the pivot threshold increased, say to 0.1.

Using only the size of the matrix as an upper bound on m_{ij} and Barlow's bound, the user can estimate the size of the matrix error terms e_{ij} using the bound of Erisman and Reid. **spRoundoff**() (p. 75) computes a tighter bound (with more work) based on work by Gear [3], $|e_{ij}| < 1.01u\rho(tc^3 + (1+t)c^2)$ where t is the threshold and c is the maximum number of off-diagonal elements in any row of L. The expensive part of computing this bound is determining the maximum number of off-diagonals in L, which changes only when the order of the matrix changes. This number is computed and saved, and only recomputed if the matrix is reordered.

[1] A. M. Erisman, J. K. Reid. Monitoring the stability of the triangular factorization of a sparse matrix. Numerische Mathematik. Vol. 22, No. 3, 1974, pp 183-186.

[2] J. L. Barlow. A note on monitoring the stability of triangular decomposition of sparse matrices. "SIAM Journal of Scientific and Statistical Computing." Vol. 7, No. 1, January 1986, pp 166-168.

[3] I. S. Duff, A. M. Erisman, J. K. Reid. "Direct Methods for Sparse Matrices." Oxford 1986. pp 99.

Returns :

If matrix is not factored, returns the magnitude of the largest element in the matrix. If the matrix is factored, a bound on the magnitude of the largest element in any of the reduced submatrices is returned.

Parameters:

eMatrix Pointer to the matrix.

4.9.2.6 void spMNA_Preorder (spMatrix *eMatrix*)

This routine massages modified node admittance matrices to remove zeros from the diagonal. It takes advantage of the fact that the row and column associated with a zero diagonal usually have structural ones placed symmetricly. This routine should be used only on modified node admittance matrices and should be executed after the matrix has been built but before the factorization begins. It should be executed for the initial factorization only and should be executed before the rows have been linked. Thus it should be run before using **spScale**() (p. 75), **spMultiply**() (p. 73), **spDeleteRowAndCol**() (p. 70), or **spNorm**() (p. 74).

This routine exploits the fact that the structural ones are placed in the matrix in symmetric twins. For example, the stamps for grounded and a floating voltage sources are

```
grounded:              floating:
[  x   x   1 ]         [  x   x   1 ]
[  x   x     ]         [  x   x  -1 ]
[  1         ]         [  1  -1     ]
```

Notice for the grounded source, there is one set of twins, and for the floating, there are two sets. We remove the zero from the diagonal by swapping the rows associated with a set of twins. For example:

```
grounded:              floating 1:            floating 2:
[  1         ]         [  1  -1     ]         [  x   x   1 ]
[  x   x     ]         [  x   x  -1 ]         [  1  -1     ]
[  x   x   1 ]         [  x   x   1 ]         [  x   x  -1 ]
```

It is important to deal with any zero diagonals that only have one set of twins before dealing with those that have more than one because swapping row destroys the symmetry of any twins in the rows being swapped, which may limit future moves. Consider

```
[  x   x   1     ]
[  x   x  -1   1 ]
[  1  -1         ]
[      1         ]
```

There is one set of twins for diagonal 4 and two for diagonal 3. Dealing with diagonal 4 first requires swapping rows 2 and 4.

```
[  x   x   1     ]
[      1         ]
[  1  -1         ]
[  x   x  -1   1 ]
```

We can now deal with diagonal 3 by swapping rows 1 and 3.

```
[  1  -1         ]
[      1         ]
[  x   x   1     ]
[  x   x  -1   1 ]
```

And we are done, there are no zeros left on the diagonal. However, if we originally dealt with diagonal 3 first, we could swap rows 2 and 3

```
[  x   x   1     ]
[  1  -1         ]
[  x   x  -1   1 ]
[      1         ]
```

Diagonal 4 no longer has a symmetric twin and we cannot continue.

So we always take care of lone twins first. When none remain, we choose arbitrarily a set of twins for a diagonal with more than one set and swap the rows corresponding to that twin. We then deal with any lone twins that were created and repeat the procedure until no zero diagonals with symmetric twins remain.

In this particular implementation, columns are swapped rather than rows. The algorithm used in this function was developed by Ken Kundert and Tom Quarles.

Parameters:

eMatrix Pointer to the matrix to be preordered.

4.9.2.7 void spMultiply (spMatrix *eMatrix*, spREAL *RHS*[], spREAL *Solution*[])

Multiplies matrix by solution vector to find source vector. Assumes matrix has not been factored. This routine can be used as a test to see if solutions are correct. It should not be used before **spMNA_Preorder()** (p. 72).

Parameters:

eMatrix Pointer to the matrix.

RHS RHS is the right hand side. This is what is being solved for.

Solution Solution is the vector being multiplied by the matrix.

iRHS iRHS is the imaginary portion of the right hand side. This is what is being solved for. This is only necessary if the matrix is complex and *spSEPARATED_COMPLEX_VECTORS* is true.

iSolution iSolution is the imaginary portion of the vector being multiplied by the matrix. This is only necessary if the matrix is complex and *spSEPARATED_COMPLEX_VECTORS* is true.

4.9.2.8 void spMultTransposed (spMatrix *eMatrix*, spREAL *RHS*[], spREAL *Solution*[])

Multiplies transposed matrix by solution vector to find source vector. Assumes matrix has not been factored. This routine can be used as a test to see if solutions are correct. It should not be used before **spMNA_Preorder**() (p. 72).

Parameters:

eMatrix Pointer to the matrix.

RHS RHS is the right hand side. This is what is being solved for.

Solution Solution is the vector being multiplied by the matrix.

iRHS iRHS is the imaginary portion of the right hand side. This is what is being solved for. This is only necessary if the matrix is complex and *spSEPARATED_COMPLEX_VECTORS* is true.

iSolution iSolution is the imaginary portion of the vector being multiplied by the matrix. This is only necessary if the matrix is complex and *spSEPARATED_COMPLEX_VECTORS* is true.

4.9.2.9 spREAL spNorm (spMatrix *eMatrix*)

Computes the L-infinity norm of an unfactored matrix. It is a fatal error to pass this routine a factored matrix.

Returns :

The largest absolute row sum of matrix.

Parameters:

eMatrix Pointer to the matrix.

4.9.2.10 spREAL spPseudoCondition (spMatrix *eMatrix*)

Computes the magnitude of the ratio of the largest to the smallest pivots. This quantity is an indicator of ill-conditioning in the matrix. If this ratio is large, and if the matrix is scaled such that uncertainties in the RHS and the matrix entries are equilibrated, then the matrix is ill-conditioned. However, a small ratio does not necessarily imply that the matrix is well-conditioned. This routine must only be used after a matrix has been factored by **spOrderAndFactor**() (p. 24) or **spFactor**() (p. 23) and before it is cleared by **spClear**() (p. 11) or **spInitialize**() (p. 14). The pseudocondition is faster to compute than the condition number calculated by **spCondition**() (p. 69), but is not as informative.

Returns :

The magnitude of the ratio of the largest to smallest pivot used during previous factorization. If the matrix was singular, zero is returned.

Parameters:

eMatrix Pointer to the matrix.

4.9.2.11 spREAL spRoundoff (spMatrix *eMatrix*, spREAL *Rho*)

This routine, along with **spLargestElement**() (p. 71), are used to gauge the stability of a factorization. See description of **spLargestElement**() (p. 71) for more information.

Returns :

Returns a bound on the magnitude of the largest element in $E = A - LU$.

Parameters:

eMatrix Pointer to the matrix.

Rho The bound on the magnitude of the largest element in any of the reduced submatrices. This is the number computed by the function **spLargestElement**() (p. 71) when given a factored matrix. If this number is negative, the bound will be computed automatically.

4.9.2.12 void spScale (spMatrix *eMatrix*, spREAL *RHS_ScaleFactors*[ ], spREAL *SolutionScaleFactors*[ ])

This function scales the matrix to enhance the possibility of finding a good pivoting order. Note that scaling enhances accuracy of the solution only if it affects the pivoting order, so it makes no sense to scale the matrix before **spFactor**() (p. 23). If scaling is desired it should be done before **spOrderAndFactor**() (p. 24). There are several things to take into account when choosing the scale factors. First, the scale factors are directly multiplied against the elements in the matrix. To prevent roundoff, each scale factor should be equal to an integer power of the number base of the machine. Since most machines operate in base two, scale factors should be a power of two. Second, the matrix should be scaled such that the matrix of element uncertainties is equilibrated. Third, this function multiplies the scale factors by the elements, so if one row tends to have uncertainties 1000 times smaller than the other rows, then its scale factor should be 1024, not 1/1024. Fourth, to save time, this function does not scale rows or columns if their scale factors are equal to one. Thus, the scale factors should be normalized to the most common scale factor. Rows and columns should be normalized separately. For example, if the size of the matrix is 100 and 10 rows tend to have uncertainties near 1e-6 and the remaining 90 have uncertainties near 1e-12, then the scale factor for the 10 should be 1/1,048,576 and the scale factors for the remaining 90 should be 1. Fifth, since this routine directly operates on the matrix, it is necessary to apply the scale factors to the RHS and Solution vectors. It may be easier to simply use **spOrderAndFactor**() (p. 24) on a scaled matrix to choose the pivoting order, and then throw away the matrix. Subsequent factorizations, performed with **spFactor**() (p. 23), will not need to have the RHS and Solution vectors descaled. Lastly, this function should not be executed before the function **spMNA_Preorder**() (p. 72).

Parameters:

eMatrix Pointer to the matrix to be scaled.

SolutionScaleFactors The array of Solution scale factors. These factors scale the columns. All scale factors are real valued.

RHS_ScaleFactors The array of RHS scale factors. These factors scale the rows. All scale factors are real valued.

4.9.2.13 void spStripFills (spMatrix *eMatrix*)

Strips the matrix of all fill-ins.

Parameters:

eMatrix Pointer to the matrix to be stripped.