Documentation for RnSparse.h and RnSparse.c

Steve Andrews

2025

1 Header file: RnSparse.h

```c
/* Steven Andrews 3/26/25.
See documentation called RnSparse_doc.tex.
Copyright 2025 by Steven Andrews.  This work is distributed under the terms
of the Gnu Lesser General Public License (LGPL). */


#ifndef __RnSparse_h
#define __RnSparse_h

#ifdef __cplusplus
extern "C" {
#endif

enum sparse_type {band};

typedef struct sparsematrixstruct{
  enum sparse_type type;     // matrix shape
  int maxrows;               // allocated matrix rows
  int nrows;                 // used matrix rows
  int fcols;                 // columns in full matrix
  int ccols;                 // columns in compressed matrix
  int *col0;                 // first column of compressed matrix
  int *col1;                 // 1+ last column of compressed matrix
  double *matrix; } *sparsematrix;


#include <stdlib.h>
#include <stdio.h>

#define sparseReadM(a,i,j) (a->matrix[(a->ccols)*(i)+(j)-(a->col0[i])])
#define sparseWriteM(a,i,j,x) ((a->matrix[(a->ccols)*(i)+(j)-(a->col0[i])])=(x))
#define sparseAddToM(a,i,j,x) ((a->matrix[(a->ccols)*(i)+(j)-(a->col0[i])])+=(x))
#define sparseMultiplyBy(a,i,j,x) ((a->matrix[(a->ccols)*(i)+(j)-(a->col0[i])])*=(
    x))

sparsematrix sparseAllocBandM(sparsematrix sprs,int maxrows,int bandsize);
void sparseFreeM(sparsematrix sprs);
void sparsePrintM(const sparsematrix sprs,int full);
void sparseSetSizeM(sparsematrix sprs,int nrows,int fcols);
void sparseClearM(sparsematrix sprs);
void sparseMultiplyMV(const sparsematrix sprs,const double *vect,double *ans);
void sparseAddIdentityM(sparsematrix sprs,double factor);
void sparseMultiplyScalarM(sparsematrix sprs,double scalar);

#ifdef __cplusplus
}
#endif

#endif
```

2 Description

These functions perform matrix operations on sparse matrices. Essentially no error checking is performed in these routines because they are designed to be as fast as possible. In particular, if the macro functions are used with row numbers or column numbers that are outside of the elements that are allowed to be non-zero, as defined by the `sparseAlloc` function, then they will simply access incorrect elements or will create a segfault error.

At present, sparse matrices are represented by a rectangular array, which works well for band diagonal matrices. However, I might change this at some point to a vector of vectors, which is better for other shapes. This change won't affect anything outside of this library, or even most code within the library, so long as matrix elements are accessed exclusively through the macros provided here.

3 Dependencies

Standard C library only.

4 History

3/26/2025 Started.

5 Math

Sparse matrices are represented internally with a matrix that has the same number of rows as the full matrix, but often compressed to fewer columns. For example, here is a band-diagonal matrix on the left and its compressed representation on the right.

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \rightarrow \begin{bmatrix} x & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & x \end{bmatrix}$$

`col0`	`col1`
−1	2
0	3
1	4
2	4

Here, `nrows=4`, which is the number of rows for both the full and compressed versions, `fcols=4`, which is the number of columns in the full matrix, and `ccols=3`, which is the number of columns for the compressed matrix. The table on the right gives the `col0` and `col1` values for each row. The `col0` value represents the column number from the full matrix that corresponds to the first column of the compressed matrix, and the `col1` value represents one more than the last column of the full matrix that is read. The x values in the diagram represent values that are stored but have no corresponding values in the full matrix. They are initialized to zero, and any operation on them (which is allowed because this can be more efficient than avoiding them) should leave them as zero.

Sparse matrices can also be stored as the full size version, but still be more efficient than non-sparse matrix code. The reason is that the `col0` and `col1` elements tell the functions where to start and stop computing, so substantial savings are still possible.

The current code is designed for band diagonal matrices. It could also be rewritten slightly for more general sparse matrices. I would do this by changing the `matrix` element of the data structure from a vector to a vector of vectors (i.e. from `double*` to `double**`).

6 Code documentation

6.1 Macros

Basic matrix reading, writing, and arithmetic functions are coded as macros for computational efficiency. They use element indices from the full matrix and convert from there. These macros are used extensively in the library functions in order to keep the code as simple as possible.

`#define sparseReadM(a,i,j) (a->matrix[(a->ccols)*(i)+(j)-(a->col0[i])])`
Reads element (i, j) from sparse matrix `a`.

`#define sparseWriteM(a,i,j,x) ((a->matrix[(a->ccols)*(i)+(j)-(a->col0[i])])=(x))`
Writes element (i, j) of sparse matrix `a` to `x`.

`#define sparseAddToM(a,i,j,x) ((a->matrix[(a->ccols)*(i)+(j)-(a->col0[i])])+=(x))`
Adds `x` to element (i, j) of sparse matrix `a`.

`#define sparseMultiplyBy(a,i,j,x)`
`((a->matrix[(a->ccols)*(i)+(j)-(a->col0[i])])*=(x))`
Multiplies element (i, j) of sparse matrix `a` by the scalar `x`.

6.2 Memory allocation and freeing

`sparsematrix sparseAllocBandM(sparsematrix sprs, int maxrows, int bandsize, int BiCGSTAB)`
Allocates or resizes a square sparse matrix with a band structure. Enter `sprs` as a pre-existing matrix if it should be resized, or as `NULL` for a new matrix. The allocated size of the corresponding full matrix is `maxrows` by `maxrows`. The functional size is also initialized to `nrows=maxrows` and `fcols=maxrows`, which can be reduced later with the `sparseSetSizeM` function. The matrix is assumed to be entirely zero, except for a band along the main diagonal that is `bandsize` wide on each side of the main diagonal, not including that diagonal. That is, the full bandwidth, including the main diagonal, is $2 \times bandsize + 1$. If an existing matrix is reallocated by calling this function an additional time but with a different `maxrows` value, then this function does *not* keep the existing data, but instead fills the entire matrix with zeros. Permissable row indices are $i \in [0, nrow - 1]$ and column indices are $j \in [i - bandsize, i + bandsize]$.

Enter `BiCGSTAB` as 1 to allocate the additional memory that is required for the BiCGSTAB functions, and as 0 to not allocate that memory. If this is done at a different time from the initial call to this function, enter `maxrows` and `bandsize` as -1, to indicate that they aren't being changed.

`void sparseFreeM(sparsematrix sprs)`
Frees all components of a sparse matrix and the data structure.

6.3 Matrix output

`void sparsePrintM(const sparsematrix sprs,int full)`
Prints a sparse matrix to the standard output. Enter `full` as 1 if the corresponding full matrix should be printed out and as 0 if only the compressed matrix should be printed. Printing shows all data structure elements, and indicates the start and end of the compresed matrix with colons. This function is primarily for debugging purposes.

6.4 Matrix manipulation

`void sparseSetSizeM(sparsematrix sprs,int nrows,int fcols)`
Sets the number of functional rows to `nrows` and the number of columns for the corresponding full matrix to `fcols`.

`void sparseClearM(sparsematrix sprs)`
Erases all data in the matrix and replaces it by zeros.

`void sparseMultiplyMV(const sparsematrix sprs,const double *vect,double *ans)`
Multiplies sparse matrix `sprs` by column vector `vect`, putting the answer in column vector `ans`.

`void sparseAddIdentityM(sparsematrix sprs,double factor)`
Adds `factor` times the identity matrix to `sprs` (the identity matrix has the same number of rows as `sprs->nrows`).

`void sparseMultiplyScalarM(sparsematrix sprs,double scalar)`
Multiplies every element of sparse matrix `sprs` by `scalar` and puts the result back in `sprs`.

`void sparseBiCGSTAB(const sparsematrix sprs, const double *b, double *x, double tolerance)`
Solves the matrix equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ for $\boldsymbol{x}$ with known $\boldsymbol{A}$, in `sprs`, and $\boldsymbol{b}$, in `bvect`. This requires an initial guess, $\boldsymbol{x}_0$, which is entered in `x`. The result is also returned in `x`. This function iterates until the error is less than `tolerance`.

The math is from Wikipedia "Biconjugate gradient stabilized method". It is as follows.

$$
\begin{aligned}
&\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0 \\
&\hat{\boldsymbol{r}}_0 = \boldsymbol{r}_0 \quad \text{there are also other options, but the code uses this} \\
&\rho_0 = (\hat{\boldsymbol{r}}_0, \boldsymbol{r}_0) \\
&\boldsymbol{p}_0 = \boldsymbol{r}_0 \\
&\quad \text{For } i = 1, 2, 3... \\
&\boldsymbol{\nu} = \boldsymbol{A} \cdot \boldsymbol{p}_{i-1} \\
&\alpha = \rho_{i-1} / (\hat{\boldsymbol{r}}_0, \boldsymbol{\nu}) \\
&\boldsymbol{h} = \boldsymbol{x}_{i-1} + \alpha \boldsymbol{p}_{i-1} \\
&\boldsymbol{s} = \boldsymbol{r}_{i-1} - \alpha \boldsymbol{\nu} \\
&\quad \text{if } \boldsymbol{s} \text{ is small enough, set } \boldsymbol{x}_i = \boldsymbol{h} \text{ and quit} \\
&\boldsymbol{t} = \boldsymbol{A} \cdot \boldsymbol{s} \\
&\omega = (\boldsymbol{t}, \boldsymbol{s}) / (\boldsymbol{t}, \boldsymbol{t}) \\
&\boldsymbol{x}_i = \boldsymbol{h} + \omega \boldsymbol{s} \\
&\boldsymbol{r}_i = \boldsymbol{s} - \omega \boldsymbol{t} \\
&\quad \text{if } \boldsymbol{r}_i \text{ is small enough, then quit} \\
&\rho_i = (\hat{\boldsymbol{r}}_0, \boldsymbol{r}_i) \\
&\beta = (\rho_i / \rho_{i-1})(\alpha / \omega) \\
&\boldsymbol{p}_i = \boldsymbol{r}_i + \beta(\boldsymbol{p}_{i-1} - \omega \boldsymbol{\nu})
\end{aligned}
$$