# Chapter 2
# Basic algorithms

A sparse matrix is one whose entries are mostly zero. There are many ways of storing a sparse matrix. Whichever method is chosen, some form of compact data structure is required that avoids storing the numerically zero entries in the matrix. It needs to be simple and flexible so that it can be used in a wide range of matrix operations. This need is met by the primary data structure in CSparse, a compressed-column matrix. Basic matrix operations that operate on this data structure are presented below, including matrix-vector multiplication, matrix-matrix multiplication, matrix addition, and transpose.

## 2.1 Sparse matrix data structures

The simplest sparse matrix data structure is a list of the nonzero entries in arbitrary order. The list consists of two integer arrays `i` and `j` and one real array `x` of length equal to the number of entries in the matrix. For example, the matrix

$$A = \begin{bmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 0.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{bmatrix} \tag{2.1}$$

is represented in *zero-based triplet* form below. A zero-based data structure for an `m`-by-`n` matrix contains row and column indices in the range `0` to `m-1` and `n-1`, respectively. A *one-based* data structure has row and column indices that start with one. The one-based convention is used in linear algebra and is presented to the MATLAB user. Internally in MATLAB and also in CSparse, all algorithms and data structures are zero-based. Thus, both conventions are used in this book, depending on the context. In particular, all C code is zero-based. All MATLAB expressions, and all linear algebraic expressions, are one-based. All pseudocode is zero-based, since it closely relates to a corresponding C code. Graph examples are one-based, since they usually relate to an example matrix (which are also one-based).

```
int i [ ]    = { 2,   1,   3,   0,   1,   3,   3,   1,   0,   2   } ;
int j [ ]    = { 2,   0,   3,   2,   1,   0,   1,   3,   0,   1   } ;
double x [ ] = { 3.0, 3.1, 1.0, 3.2, 2.9, 3.5, 0.4, 0.9, 4.5, 1.7 } ;
```

The triplet form is simple to create but difficult to use in most sparse matrix algorithms. The *compressed-column* form is more useful and is used in almost all functions in CSparse. An `m`-by-`n` sparse matrix that can contain up to `nzmax` entries is represented with an integer array `p` of length `n+1`, an integer array `i` of length `nzmax`, and a real array `x` of length `nzmax`. Row indices of entries in column `j` are stored in `i[p[j]]` through `i[p[j+1]-1]`, and the corresponding numerical values are stored in the same locations in `x`. The first entry `p[0]` is always zero, and `p[n]` $\leq$ `nzmax` is the number of actual entries in the matrix. The example matrix (2.1) is represented as

```
int p [ ]    = { 0,             3,             6,        8,       10 } ;
int i [ ]    = { 0,   1,   3,   1,   2,   3,   0,   2,   1,   3   } ;
double x [ ] = { 4.5, 3.1, 3.5, 2.9, 1.7, 0.4, 3.2, 3.0, 0.9, 1.0 } ;
```

MATLAB uses a compressed-column data structure much like `cs` for its sparse matrices. It requires the row indices in each column to appear in ascending order, and no zero entries may be present. Those two restrictions are relaxed in CSparse. The triplet form and the compressed-column data structures are both encapsulated in the `cs` structure:

```
typedef struct cs_sparse    /* matrix in compressed-column or triplet form */
{
    int nzmax ;     /* maximum number of entries */
    int m ;         /* number of rows */
    int n ;         /* number of columns */
    int *p ;        /* column pointers (size n+1) or col indices (size nzmax) */
    int *i ;        /* row indices, size nzmax */
    double *x ;     /* numerical values, size nzmax */
    int nz ;        /* # of entries in triplet matrix, -1 for compressed-col */
} cs ;
```

The array `p` contains the column pointers for the compressed-column form (of size `n+1`) or the column indices for the triplet form (of size `nzmax`). The matrix is in compressed-column form if `nz` is negative. Any given CSparse function expects its sparse matrix input in one form or the other, except for `cs_print`, `cs_spalloc`, `cs_spfree`, and `cs_sprealloc`, which can operate on either form.

Within a mexFunction written in C or Fortran (but callable from MATLAB), several functions are available that extract the parts of a MATLAB sparse matrix; `mxGetJc` returns a pointer to the equivalent of the `A->p` column pointer array of the `cs` matrix `A`. The functions `mxGetIr`, `mxGetPr`, `mxGetM`, `mxGetN`, and `mxGetNzmax` return `A->i`, `A->x`, `A->m`, `A->n`, and `A->nzmax`, respectively. These `mx` functions are not available to a MATLAB statement typed in the MATLAB command window or in a MATLAB M-file but only in a compiled C or Fortran mexFunction. The compressed-column data structures used in MATLAB and CSparse are identical, except that MATLAB can handle complex matrices as well. MATLAB 7.2 forbids explicit zero entries and requires row indices to be in order in each column.

Access to a column of `A` is simple, equivalent to `c=A(:,j)` in MATLAB, where `j` is a scalar. This assignment takes $O(|\texttt{c}|)$ time in MATLAB, which is optimal. Accessing the rows of a sparse matrix in `cs` form, or in MATLAB, is difficult. The MATLAB statement `r=A(i,:)` for a scalar `i` accesses a row of `A`. To implement this, MATLAB must examine every column of `A`, looking for row index `i` in each column. This is costly compared with accessing a column. Transposing a sparse matrix and accessing its columns is better than repeatedly accessing its rows.

The `cs` data structure can contain numerically zero entries, which brings up the important practical and theoretical issue of numerical cancellation. Exact numerical cancellation is rare, and most algorithms ignore it. An entry in the data structure that is computed but found to be numerically zero is still called a "nonzero" in this book. Leaving these entries in the matrix leads to much simpler algorithms and more elegant graph theoretical statements about the algorithms, in particular matrix-matrix multiplication, factorization, and the solution of $Lx = b$ when $b$ is sparse. Zero entries can always be dropped afterward (see Section 2.7); this is what MATLAB does. Modifying the nonzero pattern of a compressed-column matrix is not trivial. Deleting or adding single entries can take $O(|A|)$ time, since no gaps can appear between columns. For example, to delete the first entry in a matrix requires that all other entries be shifted up by one position. The MATLAB statements `A(1,1)=0 ; A(1,1)=1` are very costly because MATLAB always removes zero entries whenever they occur.

A *numerically rank-deficient* matrix is rank deficient in the usual sense. The *structural rank* of a matrix is the largest rank that can be obtained by reassigning the numerical values of the entries in its data structure. An $m$-by-$n$ matrix is *structurally rank deficient* if its structural rank is less than $\min(m, n)$. For example, $A$ is numerically rank deficient but has structural full rank, while $C$ is both numerically and structurally rank deficient:

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}.$$

## 2.2 Matrix-vector multiplication

One of the simplest sparse matrix algorithms is matrix-vector multiplication, $z = Ax + y$, where $y$ and $x$ are dense vectors and $A$ is sparse. If $A$ is split into $n$ column vectors, the result $z = Ax + y$ is

$$z = \begin{bmatrix} A_{*1} & \dots & A_{*n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} + y.$$

Allowing the result to overwrite the input vector $y$, the $j$th iteration computes $y = y + A_{*j}x_j$. The pseudocode for computing $y = Ax + y$ is given below.

**for** $j = 0$ to $n - 1$ **do**
  **for each** $i$ for which $a_{ij} \neq 0$ **do**
    $y_i = y_i + a_{ij}x_j$

Most algorithms are presented here directly in C, since the pseudocode directly translates into C with little modification. Below is the complete C version of the algorithm. Note how the `for (p = ...)` loop in the `cs_gaxpy` function takes the place of the **for each** $i$ loop in the pseudocode (the name is short for generalized $A$ times $x$ plus $y$). The MATLAB equivalent of `cs_gaxpy(A,x,y)` is `y=A*x+y`. Detailed descriptions of the inputs, outputs, and return values of all CSparse functions are given in Chapter 9.

```
int cs_gaxpy (const cs *A, const double *x, double *y)
{
    int p, j, n, *Ap, *Ai ;
    double *Ax ;
    if (!CS_CSC (A) || !x || !y) return (0) ;       /* check inputs */
    n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    for (j = 0 ; j < n ; j++)
    {
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            y [Ai [p]] += Ax [p] * x [j] ;
        }
    }
    return (1) ;
}

#define CS_CSC(A) (A && (A->nz == -1))
#define CS_TRIPLET(A) (A && (A->nz >= 0))
```

The function first checks its inputs to ensure they exist, and returns false (zero) if they do not. This protects against a caller that ran out of memory. `CS_CSC(A)` is true for a compressed-column matrix; `CS_TRIPLET(A)` is true for a matrix in triplet form. The next line (`n=A->n ; ...`) extracts the contents of the matrix `A`—its dimension, column pointers, row indices, and numerical values.

## 2.3 Utilities

A sparse matrix algorithm such as `cs_gaxpy` requires a sparse matrix in `cs` form as input. A few utility functions are required to create this data structure. The `cs_malloc`, `cs_calloc`, `cs_realloc`, and `cs_free` functions are simple wrappers around the equivalent ANSI C or MATLAB memory management functions.

```
void *cs_malloc (int n, size_t size)
{
    return (malloc (CS_MAX (n,1) * size)) ;
}

void *cs_calloc (int n, size_t size)
{
    return (calloc (CS_MAX (n,1), size)) ;
}

void *cs_free (void *p)
{
    if (p) free (p) ;       /* free p if it is not already NULL */
    return (NULL) ;         /* return NULL to simplify the use of cs_free */
}
```

`cs_realloc` changes the size of a block of memory. If successful, it returns a pointer to a block of memory of size equal to `n*size`, and sets `ok` to true. If it fails, it returns the original pointer `p` and sets `ok` to false.

```
void *cs_realloc (void *p, int n, size_t size, int *ok)
{
    void *pnew ;
    pnew = realloc (p, CS_MAX (n,1) * size) ; /* realloc the block */
    *ok = (pnew != NULL) ;                  /* realloc fails if pnew is NULL */
    return ((*ok) ? pnew : p) ;             /* return original p if failure */
}
```

The `cs_spalloc` function creates an `m`-by-`n` sparse matrix that can hold up to `nzmax` entries. Numerical values are allocated if `values` is true. A triplet or compressed-column matrix is allocated depending on whether `triplet` is true or false. `cs_spfree` frees a sparse matrix, and `cs_sprealloc` changes the maximum number of entries that a `cs` sparse matrix can contain (either triplet or compressed-column).

```
cs *cs_spalloc (int m, int n, int nzmax, int values, int triplet)
{
    cs *A = cs_calloc (1, sizeof (cs)) ;    /* allocate the cs struct */
    if (!A) return (NULL) ;                 /* out of memory */
    A->m = m ;                              /* define dimensions and nzmax */
    A->n = n ;
    A->nzmax = nzmax = CS_MAX (nzmax, 1) ;
    A->nz = triplet ? 0 : -1 ;              /* allocate triplet or comp.col */
    A->p = cs_malloc (triplet ? nzmax : n+1, sizeof (int)) ;
    A->i = cs_malloc (nzmax, sizeof (int)) ;
    A->x = values ? cs_malloc (nzmax, sizeof (double)) : NULL ;
    return ((!A->p || !A->i || (values && !A->x)) ? cs_spfree (A) : A) ;
}

cs *cs_spfree (cs *A)
{
    if (!A) return (NULL) ;     /* do nothing if A already NULL */
    cs_free (A->p) ;
    cs_free (A->i) ;
    cs_free (A->x) ;
    return (cs_free (A)) ;      /* free the cs struct and return NULL */
}

int cs_sprealloc (cs *A, int nzmax)
{
    int ok, oki, okj = 1, okx = 1 ;
    if (!A) return (0) ;
    if (nzmax <= 0) nzmax = (CS_CSC (A)) ? (A->p [A->n]) : A->nz ;
    A->i = cs_realloc (A->i, nzmax, sizeof (int), &oki) ;
    if (CS_TRIPLET (A)) A->p = cs_realloc (A->p, nzmax, sizeof (int), &okj) ;
    if (A->x) A->x = cs_realloc (A->x, nzmax, sizeof (double), &okx) ;
    ok = (oki && okj && okx) ;
    if (ok) A->nzmax = nzmax ;
    return (ok) ;
}
```

MATLAB provides similar utilities. `cs_spalloc(m,n,nzmax,1,0)` is identical to the MATLAB `spalloc(m,n,nzmax)`, and `cs_spfree(A)` is the same as `clear A`. The number of nonzeros in a compressed-column `cs` matrix `A` is given by `A->p[A->n]`, the last column pointer value; this is identical to `nnz(A)` in MATLAB if the `cs` matrix `A` has no explicit zeros. The MATLAB statement `nzmax(A)` is the same as `A->nzmax`.

## 2.4 Triplet form

The utility functions can allocate space for a sparse matrix, but they do not define its contents. The simplest way to construct a `cs` matrix is to first allocate a matrix in triplet form. Applications would normally create a matrix in this way, rather than statically defining them as done in Section 2.1. For example,

```
cs *T ;
int *Ti, *Tj ;
double *Tx ;
T = cs_spalloc (m, n, nz, 1, 1) ;
Ti = T->i ; Tj = T->p ; Tx = T->x ;
```

Next, place each entry of the sparse matrix in the `Ti`, `Tj`, and `Tx` arrays. The `k`th entry has row index $i =$ `Ti[k]`, column index $j =$ `Tj[k]`, and numerical value $a_{ij} =$ `Tx[k]`. The entries can appear in arbitrary order. Set `T->nz` to be the number of entries in the matrix. Section 2.1 gives an example of a matrix in triplet form. If multiple entries with identical row and column indices exist, the corresponding numerical value is the sum of all such *duplicate* entries.

The `cs_entry` function is useful if the number of entries in the matrix is not known when the matrix is first allocated. If space is not sufficient for the next entry, the size of the `T->i`, `T->j`, and `T->x` arrays is doubled. The dimensions of `T` are increased as needed.

```
int cs_entry (cs *T, int i, int j, double x)
{
    if (!CS_TRIPLET (T) || i < 0 || j < 0) return (0) ;     /* check inputs */
    if (T->nz >= T->nzmax && !cs_sprealloc (T,2*(T->nzmax))) return (0) ;
    if (T->x) T->x [T->nz] = x ;
    T->i [T->nz] = i ;
    T->p [T->nz++] = j ;
    T->m = CS_MAX (T->m, i+1) ;
    T->n = CS_MAX (T->n, j+1) ;
    return (1) ;
}
```

The `cs_compress` function converts this triplet-form `T` into a compressed-column matrix `C`. First, `C` and a size-`n` workspace are allocated. Next, the number of entries in each column of `C` is computed, and the column pointer array `Cp` is constructed as the cumulative sum of the column counts. The counts in `w` are also replaced with a copy of `Cp`. `cs_compress` iterates through each entry in the triplet matrix. The column pointer `w[Tj[k]]` is found and postincremented. This determines the location `p` where the row index `Ti[k]` and numerical value `Tx[k]` are placed in `C`. Finally, the workspace is freed and the result `C` is returned.

```
cs *cs_compress (const cs *T)
{
    int m, n, nz, p, k, *Cp, *Ci, *w, *Ti, *Tj ;
    double *Cx, *Tx ;
    cs *C ;
    if (!CS_TRIPLET (T)) return (NULL) ;                /* check inputs */
    m = T->m ; n = T->n ; Ti = T->i ; Tj = T->p ; Tx = T->x ; nz = T->nz ;
    C = cs_spalloc (m, n, nz, Tx != NULL, 0) ;          /* allocate result */
    w = cs_calloc (n, sizeof (int)) ;                   /* get workspace */
    if (!C || !w) return (cs_done (C, w, NULL, 0)) ;    /* out of memory */
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    for (k = 0 ; k < nz ; k++) w [Tj [k]]++ ;           /* column counts */
    cs_cumsum (Cp, w, n) ;                              /* column pointers */
    for (k = 0 ; k < nz ; k++)
    {
        Ci [p = w [Tj [k]]++] = Ti [k] ;    /* A(i,j) is the pth entry in C */
        if (Cx) Cx [p] = Tx [k] ;
    }
    return (cs_done (C, w, NULL, 1)) ;      /* success; free w and return C */
}
```

The cs_done function returns a cs sparse matrix and frees any workspace.

```
cs *cs_done (cs *C, void *w, void *x, int ok)
{
    cs_free (w) ;                       /* free workspace */
    cs_free (x) ;
    return (ok ? C : cs_spfree (C)) ;   /* return result if OK, else free it */
}
```

Computing the cumulative sum will be useful in other CSparse functions, so it appears as its own function, cs_cumsum. It sets p[i] equal to the sum of c[0] through c[i-1]. It returns the sum of c[0...n-1]. On output, c[0...n-1] is overwritten with p[0...n-1].

```
double cs_cumsum (int *p, int *c, int n)
{
    int i, nz = 0 ;
    double nz2 = 0 ;
    if (!p || !c) return (-1) ;     /* check inputs */
    for (i = 0 ; i < n ; i++)
    {
        p [i] = nz ;
        nz += c [i] ;
        nz2 += c [i] ;              /* also in double to avoid int overflow */
        c [i] = p [i] ;             /* also copy p[0..n-1] back into c[0..n-1]*/
    }
    p [n] = nz ;
    return (nz2) ;                  /* return sum (c [0..n-1]) */
}
```

The MATLAB statement C=sparse(i,j,x,m,n) performs the same function as cs_compress, except that it returns a matrix with sorted columns, and sums up duplicate entries (see Sections 2.5 and 2.6).

## 2.5 Transpose

The algorithm for transposing a sparse matrix ($C = A^T$) is very similar to the cs_compress function because it can be viewed not just as a linear algebraic function but as a method for converting a compressed-column sparse matrix into a compressed-row sparse matrix as well. The algorithm computes the row counts of A, computes the cumulative sum to obtain the row pointers, and then iterates over each nonzero entry in A, placing the entry in its appropriate row vector. If the resulting sparse matrix C is interpreted as a matrix in compressed-row form, then C is equal to A, just in a different format. If C is viewed as a compressed-column matrix, then C contains $A^T$. It is simpler to describe cs_transpose with C as a row-oriented matrix.

```
cs *cs_transpose (const cs *A, int values)
{
    int p, q, j, *Cp, *Ci, n, m, *Ap, *Ai, *w ;
    double *Cx, *Ax ;
    cs *C ;
    if (!CS_CSC (A)) return (NULL) ;    /* check inputs */
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    C = cs_spalloc (n, m, Ap [n], values && Ax, 0) ;       /* allocate result */
    w = cs_calloc (m, sizeof (int)) ;                      /* get workspace */
    if (!C || !w) return (cs_done (C, w, NULL, 0)) ;       /* out of memory */
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    for (p = 0 ; p < Ap [n] ; p++) w [Ai [p]]++ ;          /* row counts */
    cs_cumsum (Cp, w, m) ;                                 /* row pointers */
    for (j = 0 ; j < n ; j++)
    {
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            Ci [q = w [Ai [p]]++] = j ; /* place A(i,j) as entry C(j,i) */
            if (Cx) Cx [q] = Ax [p] ;
        }
    }
    return (cs_done (C, w, NULL, 1)) ;  /* success; free w and return C */
}
```

First, the output matrix C and workspace w are allocated. Next, the row counts and their cumulative sum are computed. The cumulative sum defines the row pointer array Cp. Finally, cs_transpose traverses each column j of A, placing column index j into each row i of C for which $a_{ij}$ is nonzero. The position q of this entry in C is given by q = w[i], which is then postincremented to prepare for the next entry to be inserted into row i. Compare cs_transpose and cs_compress. Their only significant difference is what kind of data structure their inputs are in. The statement C=cs_transpose(A) is identical to the MATLAB statement C=A', except that the latter can also compute the complex conjugate transpose. For real matrices the MATLAB statements C=A' and C=A.' are identical. The values parameter is true (nonzero) to signify that the numerical values of C are to be computed or false (zero) otherwise.

Sorting the columns of a sparse matrix is particularly simple. The statement C=cs_transpose(A) computes the transpose of A. Each row of C is constructed one column index at a time, from column 0 to C->n-1. Thus, it is a sorted matrix;

`cs_transpose` is a linear-time bucket sort algorithm. `A` can be sorted by transposing it twice. A `cs_sort` function is left as an exercise. The total time required is $O(m + n + |A|)$. Rather than transposing a matrix twice, it is sometimes possible to create the transpose first and then sort it with a single call to `cs_transpose`.

MATLAB has no explicit function to sort its sparse matrices. Each function or operator that returns a sparse matrix is required to return it with sorted columns.

## 2.6 Summing up duplicate entries

Finite-element methods generate a matrix as a collection of *elements* or dense submatrices. The complete matrix is a summation of the elements. If two elements contribute to the same entry, their values should be summed. The `cs_compress` function leaves these duplicate entries in its output matrix. They can be summed with the `cs_dupl` function.

```
int cs_dupl (cs *A)
{
    int i, j, p, q, nz = 0, n, m, *Ap, *Ai, *w ;
    double *Ax ;
    if (!CS_CSC (A)) return (0) ;               /* check inputs */
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    w = cs_malloc (m, sizeof (int)) ;           /* get workspace */
    if (!w) return (0) ;                        /* out of memory */
    for (i = 0 ; i < m ; i++) w [i] = -1 ;      /* row i not yet seen */
    for (j = 0 ; j < n ; j++) loop col
    {
        q = nz ;                                /* column j will start at q */
        for (p = Ap [j] ; p < Ap [j+1] ; p++) loop row
        {
            i = Ai [p] ;                        /* A(i,j) is nonzero */
            if (w [i] >= q)
            {
                Ax [w [i]] += Ax [p] ;          /* A(i,j) is a duplicate */
            }
            else
            {
                w [i] = nz ;                    /* record where row i occurs */
                Ai [nz] = i ;                   /* keep A(i,j) */
                Ax [nz++] = Ax [p] ;
            }
        }
        Ap [j] = q ;                            /* record start of column j */
    }
    Ap [n] = nz ;                               /* finalize A */
    cs_free (w) ;                               /* free workspace */
    return (cs_sprealloc (A, 0)) ;              /* remove extra space from A */
}
```

The function uses a size-`m` integer workspace; `w[i]` records the location in `Ai` and `Ax` of the most recent entry with row index `i`. If this position is within the current column `j`, then it is a duplicate entry and must be summed. Otherwise, the entry is kept and `w[i]` is updated to reflect the position of this entry.

MATLAB does not have an explicit function to sum duplicate entries of a sparse matrix. It is combined with the MATLAB `sparse` function that converts a triplet matrix to a compressed sparse matrix.

## 2.7 Removing entries from a matrix

CSparse does not require its sparse matrices to be free of numerically zero entries, but its MATLAB interface does. Rather than writing a special-purpose function to drop zeros from a matrix, the `cs_fkeep` function is used. It takes as an argument a pointer to a function `fkeep(i,j,aij,other)` which is evaluated for each entry $a_{ij}$ in the matrix. An entry is kept if `fkeep` is true for that entry. Dropping entries from `A` requires each column to be shifted; `Ap[j]` must be decremented by the number of entries dropped from columns `0` to `j-1`. When a `cs` matrix `A` is returned to MATLAB, `cs_dropzeros(A)` is normally performed first. The `cs_chol` mexFunction optionally keeps zero entries in $L$, so that `cs_updown` can work properly.

```
int cs_fkeep (cs *A, int (*fkeep) (int, int, double, void *), void *other)
{
    int j, p, nz = 0, n, *Ap, *Ai ;
    double *Ax ;
    if (!CS_CSC (A) || !fkeep) return (-1) ;    /* check inputs */
    n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    for (j = 0 ; j < n ; j++)
    {
        p = Ap [j] ;                            /* get current location of col j */
        Ap [j] = nz ;                           /* record new location of col j */
        for ( ; p < Ap [j+1] ; p++)
        {
            if (fkeep (Ai [p], j, Ax ? Ax [p] : 1, other))
            {
                if (Ax) Ax [nz] = Ax [p] ;  /* keep A(i,j) */
                Ai [nz++] = Ai [p] ;
            }
        }
    }
    Ap [n] = nz ;                               /* finalize A */
    cs_sprealloc (A, 0) ;                       /* remove extra space from A */
    return (nz) ;
}

static int cs_nonzero (int i, int j, double aij, void *other)
{
    return (aij != 0) ;
}
int cs_dropzeros (cs *A)
{
    return (cs_fkeep (A, &cs_nonzero, NULL)) ;  /* keep all nonzero entries */
}
```

Additional arguments can be passed to `fkeep` via the `void *other` parameter to `cs_fkeep`. This is demonstrated by `cs_droptol`, which removes entries whose magnitude is less than or equal to `tol`.

The MATLAB equivalent for `cs_droptol(A,tol)` is `A = A.*(abs(A)>tol)`.

```
static int cs_tol (int i, int j, double aij, void *tol)
{
    return (fabs (aij) > *((double *) tol)) ;
}
int cs_droptol (cs *A, double tol)
{
    return (cs_fkeep (A, &cs_tol, &tol)) ;    /* keep all large entries */
}
```

## 2.8 Matrix multiplication

Since matrices are stored in compressed-column form in CSparse, the matrix multiplication $C = AB$, where $C$ is $m$-by-$n$, $A$ is $m$-by-$k$, and $B$ is $k$-by-$n$, should access $A$ and $B$ by column and create $C$ one column at a time. If $C_{*j}$ and $B_{*j}$ denote column $j$ of $C$ and $B$, then $C_{*j} = AB_{*j}$. Splitting $A$ into its $k$ columns and $B_{*j}$ into its $k$ individual entries,

$$C_{*j} = \left[ \begin{array}{ccc} A_{*1} & \cdots & A_{*k} \end{array} \right] \left[ \begin{array}{c} b_{1j} \\ \vdots \\ b_{kj} \end{array} \right] = \sum_{i=1}^{k} A_{*i} b_{ij}. \tag{2.2}$$

The nonzero pattern of $C$ is given by the following theorem.

**Theorem 2.1** (Gilbert [101]). *The nonzero pattern of $C_{*j}$ is the set union of the nonzero pattern of $A_{*i}$ for all $i$ for which $b_{ij}$ is nonzero. If $\mathcal{C}_j$, $\mathcal{A}_i$, and $\mathcal{B}_j$ denote the set of row indices of nonzero entries in $C_{*j}$, $A_{*i}$, and $B_{*j}$, then*

$$\mathcal{C}_j = \bigcup_{i \in \mathcal{B}_j} \mathcal{A}_i. \tag{2.3}$$

A matrix multiplication algorithm must compute both $C_{*j}$ and $\mathcal{C}_j$. Note that (2.3) is correct only if numerical cancellation is ignored. It is implemented with `cs_scatter` and `cs_multiply` below. A dense vector `x` is used to construct $C_{*j}$. The set $\mathcal{C}_j$ is stored directly in `C`, but another work vector `w` is needed to determine if a given row index $i$ is in the set already. The vector `w` starts out cleared. When computing column `j`, `w[i]<j+1` will denote a row index `i` that is not yet in $\mathcal{C}_j$. When `i` is inserted in $\mathcal{C}_j$, `w[i]` is set to `j+1`. The `cs_scatter` function computes one iteration of (2.2) and (2.3) for a single value of `i`, using a *scatter* operation to copy a sparse vector into a dense one. The matrix multiplication function `cs_multiply` first allocates the `w` and `x` workspace and the output matrix `C`. Next, it iterates over each column `j` of the result `C`. After a series of scatter operations, the dense vector `x` is *gathered* into a sparse vector (a column of `C`). Since the number of nonzeros in `C` is not known at the beginning, it is increased in size as needed.

Computing `nnz(A*B)` is actually much harder than computing `nnz(chol(A))`. The latter is discussed in Chapter 4. An alternate approach that computes `nnz(A*B)` in an initial pass and then `C=A*B` in a second pass is left as an exercise (Problem 2.20).

```
cs *cs_multiply (const cs *A, const cs *B)
{
    int p, j, nz = 0, anz, *Cp, *Ci, *Bp, m, n, bnz, *w, values, *Bi ;
    double *x, *Bx, *Cx ;
    cs *C ;
    if (!CS_CSC (A) || !CS_CSC (B)) return (NULL) ;      /* check inputs */
    m = A->m ; anz = A->p [A->n] ;
    n = B->n ; Bp = B->p ; Bi = B->i ; Bx = B->x ; bnz = Bp [n] ;
    w = cs_calloc (m, sizeof (int)) ;                    /* get workspace */
    values = (A->x != NULL) && (Bx != NULL) ;
    x = values ? cs_malloc (m, sizeof (double)) : NULL ; /* get workspace */
    C = cs_spalloc (m, n, anz + bnz, values, 0) ;        /* allocate result */
    if (!C || !w || (values && !x)) return (cs_done (C, w, x, 0)) ;
    Cp = C->p ;                       C_*j
    for (j = 0 ; j < n ; j++)
    {
        if (nz + m > C->nzmax && !cs_sprealloc (C, 2*(C->nzmax)+m))
        {
            return (cs_done (C, w, x, 0)) ;             /* out of memory */
        }
        Ci = C->i ; Cx = C->x ;         /* C->i and C->x may be reallocated */
        Cp [j] = nz ;                   /* column j of C starts here */
        for (p = Bp [j] ; p < Bp [j+1] ; p++)                 A_*i b_ij
        {
            nz = cs_scatter (A, Bi [p], Bx ? Bx [p] : 1, w, x, j+1, C, nz) ;
        }
        if (values) for (p = Cp [j] ; p < nz ; p++) Cx [p] = x [Ci [p]] ;
    }
    Cp [n] = nz ;                       /* finalize the last column of C */
    cs_sprealloc (C, 0) ;               /* remove extra space from C */
    return (cs_done (C, w, x, 1)) ;     /* success; free workspace, return C */
}

int cs_scatter (const cs *A, int j, double beta, int *w, double *x, int mark,
    cs *C, int nz)
{
    int i, p, *Ap, *Ai, *Ci ;
    double *Ax ;
    if (!CS_CSC (A) || !w || !CS_CSC (C)) return (-1) ;     /* check inputs */
    Ap = A->p ; Ai = A->i ; Ax = A->x ; Ci = C->i ;
    for (p = Ap [j] ; p < Ap [j+1] ; p++)
    {
        i = Ai [p] ;                            /* A(i,j) is nonzero */
        if (w [i] < mark)
        {
            w [i] = mark ;                      /* i is new entry in column j */
            Ci [nz++] = i ;                     /* add i to pattern of C(:,j) */
            if (x) x [i] = beta * Ax [p] ;      /* x(i) = beta*A(i,j) */
        }
        else if (x) x [i] += beta * Ax [p] ;    /* i exists in C(:,j) already */
    }
    return (nz) ;
}
```

When `cs_multiply` is finished, the matrix `C` is resized to the actual number of entries it contains, and the workspace is freed. The `cs_scatter` function computes `x=x+beta*A(:,j)`, and accumulates the nonzero pattern of `x` in `C->i`, starting at position nz. The new value of nz is returned. Row index i is in the pattern of x if w[i] is equal to mark.

The time taken by cs_multiply is $O(n + f + |B|)$, where $f$ is the number of floating-point operations performed ($f$ dominates the run time unless $A$ has one or more columns with no entries, in which case either $n$ or $|B|$ can be greater than $f$). If the columns of C need to be sorted, either $C = ((AB)^T)^T$ or $C = (B^T A^T)^T$ can be computed. The latter is better if $C$ has many more entries than $A$ or $B$. The MATLAB equivalent C=A*B uses a similar algorithm to the one presented here.

## 2.9 Matrix addition

The cs_add function performs matrix addition, $C = \alpha A + \beta B$. Matrix addition can be written as a multiplication of two matrices,

$$C = \begin{bmatrix} A & B \end{bmatrix} \begin{bmatrix} \alpha I \\ \beta I \end{bmatrix}, \tag{2.4}$$

where $I$ is an identity matrix of the appropriate size. Although it is not implemented this way, the function cs_add looks very much like cs_multiply because of (2.4). The innermost loop differs slightly; no reallocation is needed, and the for p loop is replaced with two calls to cs_scatter. Like cs_multiply, it does not return C with sorted columns. The MATLAB equivalent is C=alpha*A+beta*B.

```
cs *cs_add (const cs *A, const cs *B, double alpha, double beta)
{
    int p, j, nz = 0, anz, *Cp, *Ci, *Bp, m, n, bnz, *w, values ;
    double *x, *Bx, *Cx ;
    cs *C ;
    if (!CS_CSC (A) || !CS_CSC (B)) return (NULL) ;         /* check inputs */
    m = A->m ; anz = A->p [A->n] ;
    n = B->n ; Bp = B->p ; Bx = B->x ; bnz = Bp [n] ;
    w = cs_calloc (m, sizeof (int)) ;                       /* get workspace */
    values = (A->x != NULL) && (Bx != NULL) ;
    x = values ? cs_malloc (m, sizeof (double)) : NULL ;    /* get workspace */
    C = cs_spalloc (m, n, anz + bnz, values, 0) ;           /* allocate result*/
    if (!C || !w || (values && !x)) return (cs_done (C, w, x, 0)) ;
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    for (j = 0 ; j < n ; j++)
    {
        Cp [j] = nz ;                   /* column j of C starts here */
        nz = cs_scatter (A, j, alpha, w, x, j+1, C, nz) ;   /* alpha*A(:,j)*/
        nz = cs_scatter (B, j, beta, w, x, j+1, C, nz) ;    /* beta*B(:,j) */
        if (values) for (p = Cp [j] ; p < nz ; p++) Cx [p] = x [Ci [p]] ;
    }
    Cp [n] = nz ;                       /* finalize the last column of C */
    cs_sprealloc (C, 0) ;               /* remove extra space from C */
    return (cs_done (C, w, x, 1)) ;     /* success; free workspace, return C */
}
```

## 2.10 Vector permutation

An $n$-by-$n$ permutation matrix $P$ can be represented by a sparse matrix `P` with a one in each row and column, or by a length-`n` integer vector `p` called a *permutation vector*, where `p[k]=i` means that $p_{ki} = 1$. A permutation matrix $P$ is orthogonal, so its inverse is simply $P^{-1} = P^T$. The inverse permutation vector is given by `pinv[i]=k` if $p_{ki} = 1$, since this implies $(P^T)_{ik} = 1$.

Some MATLAB functions return permutation vectors; others return permutation matrices. If `p` and `q` are MATLAB permutation vectors of length `n`, converting between these forms is done as follows:

| | |
|---|---|
| `[p j x] = find(P')` | convert row permutation `P*A` to `A(p,:)` |
| `[q j x] = find(Q)` | convert column permutation `A*Q` to `A(:,q)` |
| `P=sparse(1:n, p, 1)` | convert row permutation `A(p,:)` to `P*A` |
| `Q=sparse(q, 1:n, 1)` | convert column permutation `A(:,q)` to `A*Q` |

If $x = Pb$, row `k` of $x$ is row `p[k]` of $b$. The function `cs_pvec` computes $x = Pb$, or `x=b(p)` in MATLAB, where `x` and `b` are vectors of length `n`. The function `cs_ipvec` computes $x = P^Tb$, or `x(p)=b` in MATLAB.

```
int cs_pvec (const int *p, const double *b, double *x, int n)
{
    int k ;
    if (!x || !b) return (0) ;                              /* check inputs */
    for (k = 0 ; k < n ; k++) x [k] = b [p ? p [k] : k] ;
    return (1) ;
}
```

Null?

```
int cs_ipvec (const int *p, const double *b, double *x, int n)
{
    int k ;
    if (!x || !b) return (0) ;                              /* check inputs */
    for (k = 0 ; k < n ; k++) x [p ? p [k] : k] = b [k] ;
    return (1) ;
}
```

The inverse, or transpose, of a permutation vector `p[k]=i` is the vector `pinv`, where `pinv[i]=k`. This is computed by `cs_pinv`. In MATLAB, `pinv(p) = 1:n` computes the inverse `pinv` of a permutation vector `p` of length `n` (this assumes that `pinv` is initially not defined or a vector of length `n` or less).

```
int *cs_pinv (int const *p, int n)
{
    int k, *pinv ;
    if (!p) return (NULL) ;                         /* p = NULL denotes identity */
    pinv = cs_malloc (n, sizeof (int)) ;            /* allocate result */
    if (!pinv) return (NULL) ;                      /* out of memory */
    for (k = 0 ; k < n ; k++) pinv [p [k]] = k ;/* invert the permutation */
    return (pinv) ;                                 /* return result */
}
```

## 2.11 Matrix permutation

The `cs_permute` function permutes a sparse matrix, $C = PAQ$ (`C=A(p,q)` in MATLAB). It takes as input a column permutation vector `q` of length `n` and an inverse row permutation `pinv` (not `p`) of length `m`, where `A` is `m`-by-`n`. Row `i` of `A` becomes row `k` of `C` if `pinv[i]=k`. The algorithm traverses the columns of `j` of `A` in permuted order via `q`. Each row index in `A` is mapped to its permuted row in `C`.

```
cs *cs_permute (const cs *A, const int *pinv, const int *q, int values)
{
    int t, j, k, nz = 0, m, n, *Ap, *Ai, *Cp, *Ci ;
    double *Cx, *Ax ;
    cs *C ;
    if (!CS_CSC (A)) return (NULL) ;    /* check inputs */
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    C = cs_spalloc (m, n, Ap [n], values && Ax != NULL, 0) ;  /* alloc result */
    if (!C) return (cs_done (C, NULL, NULL, 0)) ;   /* out of memory */
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    for (k = 0 ; k < n ; k++)
    {
        Cp [k] = nz ;                   /* column k of C is column q[k] of A */
        j = q ? (q [k]) : k ;
        for (t = Ap [j] ; t < Ap [j+1] ; t++)
        {
            if (Cx) Cx [nz] = Ax [t] ;  /* row i of A is row pinv[i] of C */
            Ci [nz++] = pinv ? (pinv [Ai [t]]) : Ai [t] ;
        }
    }
    Cp [n] = nz ;                       /* finalize the last column of C */
    return (cs_done (C, NULL, NULL, 1)) ;
}
```

CSparse functions that operate on symmetric matrices use just the upper triangular part, just like `chol` in MATLAB. If `A` is symmetric with only the upper triangular part stored, `C=A(p,p)` is not upper triangular. The `cs_symperm` function computes `C=A(p,p)` for a symmetric matrix `A` whose upper triangular part is stored, returning `C` in the same format. Entries below the diagonal are ignored.

The first `for j` loop counts how many entries are in each column of `C`. Suppose `i` $\leq$ `j`, and `A(i,j)` is permuted to become entry `C(i2,j2)`. If `i2` $\leq$ `j2`, this entry is in the upper triangular part of `C`. Otherwise, `C(i2,j2)` is in the lower triangular part of `C`, and the entry must be placed in `C` as `C(j2,i2)` instead. After the column counts of `C` are computed (in `w`), the cumulative sum is computed to obtain the column pointers `Cp`. The second `for` loop constructs `C`, much like `cs_permute`.

```
cs *cs_symperm (const cs *A, const int *pinv, int values)
{
    int i, j, p, q, i2, j2, n, *Ap, *Ai, *Cp, *Ci, *w ;
    double *Cx, *Ax ;
    cs *C ;
    if (!CS_CSC (A)) return (NULL) ;                    /* check inputs */
    n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    C = cs_spalloc (n, n, Ap [n], values && (Ax != NULL), 0) ; /* alloc result*/
    w = cs_calloc (n, sizeof (int)) ;                   /* get workspace */
    if (!C || !w) return (cs_done (C, w, NULL, 0)) ;    /* out of memory */
```

```
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    for (j = 0 ; j < n ; j++)           /* count entries in each column of C */
    {
        j2 = pinv ? pinv [j] : j ;      /* column j of A is column j2 of C */
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            i = Ai [p] ;
            if (i > j) continue ;       /* skip lower triangular part of A */
            i2 = pinv ? pinv [i] : i ;  /* row i of A is row i2 of C */
            w [CS_MAX (i2, j2)]++ ;     /* column count of C */
        }
    }
    cs_cumsum (Cp, w, n) ;              /* compute column pointers of C */
    for (j = 0 ; j < n ; j++)
    {
        j2 = pinv ? pinv [j] : j ;      /* column j of A is column j2 of C */
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            i = Ai [p] ;
            if (i > j) continue ;       /* skip lower triangular part of A*/
            i2 = pinv ? pinv [i] : i ;  /* row i of A is row i2 of C */
            Ci [q = w [CS_MAX (i2, j2)]++] = CS_MIN (i2, j2) ;
            if (Cx) Cx [q] = Ax [p] ;
        }
    }
    return (cs_done (C, w, NULL, 1)) ;  /* success; free workspace, return C */
}
```

## 2.12 Matrix norm

Computing the 2-norm of a sparse matrix ($\|A\|_2$) is not trivial, since it is the largest singular value of $A$. MATLAB does not provide a function for computing the 2-norm of a sparse matrix, although it can compute a good estimate using `normest`. The $\infty$-norm is the maximum row-sum, the computation of which requires a workspace of size `n` if `A` is accessed by column. The simplest norm to use for a sparse matrix stored in compressed-column form is the 1-norm, $\|A\|_1 = \max_j \sum_{i=1}^m |a_{ij}|$, which is computed by the `cs_norm` function below. Note that it does not make use of the `A->i` row index array. The MATLAB `norm` function can compute the 1-norm, $\infty$-norm, or Frobenius norm of a sparse matrix.

```
double cs_norm (const cs *A)
{
    int p, j, n, *Ap ;
    double *Ax,  norm = 0, s ;
    if (!CS_CSC (A) || !A->x) return (-1) ;             /* check inputs */
    n = A->n ; Ap = A->p ; Ax = A->x ;
    for (j = 0 ; j < n ; j++)
    {
        for (s = 0, p = Ap [j] ; p < Ap [j+1] ; p++) s += fabs (Ax [p]) ;
        norm = CS_MAX (norm, s) ;
    }
    return (norm) ;
}
```

## 2.13 Reading a matrix from a file

The `cs_load` function reads in a triplet matrix from a file. The matrix `T` is initially allocated as a 0-by-0 triplet matrix with space for just one entry. The dimensions of `T` are determined by the maximum row and column index read from the file.

```
cs *cs_load (FILE *f)
{
    int i, j ;
    double x ;
    cs *T ;
    if (!f) return (NULL) ;                             /* check inputs */
    T = cs_spalloc (0, 0, 1, 1, 1) ;                    /* allocate result */
    while (fscanf (f, "%d %d %lg\n", &i, &j, &x) == 3)
    {
        if (!cs_entry (T, i, j, x)) return (cs_spfree (T)) ;
    }
    return (T) ;
}
```

## 2.14 Printing a matrix

`cs_print` prints the contents of a `cs` matrix in triplet form or compressed-column form. Only the first few entries are printed if `brief` is true.

```
int cs_print (const cs *A, int brief)
{
    int p, j, m, n, nzmax, nz, *Ap, *Ai ;
    double *Ax ;
    if (!A) { printf ("(null)\n") ; return (0) ; }
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    nzmax = A->nzmax ; nz = A->nz ;
    printf ("CSparse Version %d.%d.%d, %s.  %s\n", CS_VER, CS_SUBVER,
        CS_SUBSUB, CS_DATE, CS_COPYRIGHT) ;
    if (nz < 0)
    {
        printf ("%d-by-%d, nzmax: %d nnz: %d, 1-norm: %g\n", m, n, nzmax,
                Ap [n], cs_norm (A)) ;
        for (j = 0 ; j < n ; j++)
        {
            printf ("    col %d : locations %d to %d\n", j, Ap [j], Ap [j+1]-1);
            for (p = Ap [j] ; p < Ap [j+1] ; p++)
            {
                printf ("      %d : %g\n", Ai [p], Ax ? Ax [p] : 1) ;
                if (brief && p > 20) { printf ("  ...\n") ; return (1) ; }
            }
        }
    }
    else
    {
        printf ("triplet: %d-by-%d, nzmax: %d nnz: %d\n", m, n, nzmax, nz) ;
        for (p = 0 ; p < nz ; p++)
        {
            printf ("    %d %d : %g\n", Ai [p], Ap [p], Ax ? Ax [p] : 1) ;
            if (brief && p > 20) { printf ("  ...\n") ; return (1) ; }
        }
    }
    return (1) ;
}
```

## 2.15 Sparse matrix collections

Arbitrary random matrices are easy to generate; random sparse matrices with specific properties are not simple to generate (type the command `type sprand` in MATLAB and compare the 3-input versus 4-input usage of the function). Both can give misleading performance results. Sparse matrices from real applications are better, such as those from the Rutherford-Boeing collection[4] [55], the NIST Matrix Market,[5] and the UF Sparse Matrix Collection.[6] The UFget package distributed with CSparse provides a simple MATLAB interface to the UF Sparse Matrix Collection. For example, `UFget('HB/arc130')` downloads the `arc130` matrix and loads it into MATLAB. `UFweb('HB/arc130')` brings up a web browser with the web page for the same matrix. Matrix properties are listed in an index, which makes it simple to write a MATLAB program that uses a selected subset of matrices (for example, all symmetric positive definite matrices in order of increasing number of nonzeros). As of April 2006, the UF Sparse Matrix Collection contains 1,377 matrices, with order 5 to 5 million, and as few as 15 and as many as 99 million nonzeros. The submission of new matrices not represented by the collection is always welcome.

## 2.16 Further reading

The CHOLMOD [30] package provides some of the sparse matrix operators in MATLAB. Other sparse matrix packages have similar functions; see the HSL[7] and the BCSLIB-EXT[8] packages in particular. Gilbert, Moler, and Schreiber present the early development of sparse matrices in MATLAB [105]. Gustavson discusses sparse matrix permutation, transpose, and multiplication [121]. The Sparse BLAS [43, 44, 56, 70] includes many of these operations.

## Exercises

2.1. Write a `cs_gatxpy` function that computes $y = A^T x + y$ without forming $A^T$.

2.2. Write a function `cs_find` that converts a `cs` matrix into a triplet-form matrix, like the `find` function in MATLAB.

2.3. Write a variant of `cs_gaxpy` that computes $y = Ax + y$, where $A$ is a symmetric matrix with only the upper triangular part present. Ignore entries in the lower triangular part.

2.4. Write a function with prototype `void cs_scale(cs *A, double *r, double *c)` that overwrites $A$ with $RAC$, where $R$ and $C$ are diagonal matrices; `r[k]` and `c[k]` are the `k`th diagonal entries of $R$ and $C$, respectively.

2.5. Write a function similar to `cs_entry` that adds a dense submatrix to a triplet

[4] www.cse.clrc.ac.uk/nag/hb

[5] math.nist.gov/MatrixMarket

[6] www.cise.ufl.edu/research/sparse/matrices; see also www.siam.org/books/fa02

[7] www.cse.clrc.ac.uk/nag/hsl

[8] www.boeing.com/phantom/bcslib-ext

matrix. `i` and `j` should be integer arrays of length `k`, and `x` should be a `k`-by-`k` dense matrix.

2.6. Show how to transpose a `cs` matrix in triplet form in $O(1)$ time.

2.7. Write a function `cs_sort` that sorts a `cs` matrix. Its prototype should be `cs *cs_sort (cs *A)`. Use two calls to `cs_transpose`. Why is `C=cs_transpose (cs_transpose (A))` incorrect?

2.8. Write a function that sorts a matrix one column at a time, using the ANSI C quicksort function, `qsort`. Compare its performance (time and memory usage) with the solution to Problem 2.7.

2.9. Write a function that creates a compressed-column matrix from a triplet matrix with sorted columns, no duplicates, and no numerically zero entries.

2.10. Show how to multiply a matrix in triplet form times a dense vector.

2.11. Sorting a matrix with a double transpose does extra work that is not required. The second transpose counts the entries in each row, but these are equal to the original column counts. Write a `cs_sort` function that avoids extra work.

2.12. Write a function `cs_ok` that checks a matrix to see if it is valid and optionally prints the matrix with prototype `int cs_ok (cs *A, int sorted, int values, int print)`. If `values` is negative, `A->x` is ignored and may be `NULL`; otherwise, it must be non-`NULL`. If `sorted` is true, then the columns must be sorted. If `values` is positive, then there can be no numerically zero entries in `A`. The time and workspace are $O(m + n + |A|)$ and $O(m)$.

2.13. Write a function that determines if a sparse matrix is symmetric.

2.14. Write a function `cs *cs_copy (cs *A)` that returns a copy of `A`.

2.15. Write a function `cs_band(A,k1,k2)` that removes all entries from `A` except for those in diagonals `k1` to `k2` of `A`. Entries outside the band should be dropped. Hint: use `cs_fkeep`.

2.16. Write a function that creates a sparse matrix copy of a dense matrix stored in column-major form.

2.17. How much time does it take to transpose a column vector? How much space does a sparse row vector take if stored in compressed-column form?

2.18. How much time and space does it take to compute $x^T y$ for two sparse column vectors $x$ and $y$, using `cs_transpose` and `cs_multiply`? Write a more efficient routine with prototype `double cs_dot (cs *x, cs *y)`, which assumes `x` and `y` are column vectors. Consider two cases: (1) The row indices of `x` and `y` are not sorted. A `double` workspace `w` of size `x->m` will need to be allocated. (2) The row indices of `x` and `y` are sorted. No workspace is required. Both cases take $O(|x| + |y|)$ time.

2.19. The first call to `cs_scatter` in each iteration of the `j` loop in both `cs_multiply` and `cs_add` does more work than is necessary, since `w[i]<mark` is always true in this case. Write a more efficient version.

2.20. Consider an alternative algorithm for `cs_multiply` that uses two passes. The first pass computes the number of entries in each column of `C` (or just the total number of entries), and the second pass performs the matrix multiplication.

No `cs_sprealloc` is needed. Compare with the original `cs_multiply`.

2.21. How efficient is `cs_add` when `A` and `B` are sparse column vectors? Hint: how much time does `calloc` take? Write faster function `cs_saxpy` that takes an initialized workspace (`w` and `x`) as input, computes the result, and returns the workspace ready to use in a subsequent call to `cs_saxpy`.

2.22. Write two functions `cs_hcat` and `cs_vcat` that perform the horizontal and vertical concatenation of `A` and `B`, respectively, just like the MATLAB statements `C = [A B]` and `C = [A ; B]`.

2.23. Write a function that implements the MATLAB statement `C=A(i1:i2,j1:j2)`. This is much simpler than the next two problems.

2.24. The MATLAB statement `C=A(i,j)`, where `i` and `j` are integer vectors, creates a submatrix `C` of `A` of dimension `length(i)`-by-`length(j)`. Write a function that performs this operation. Either assume that `i` and `j` do not contain duplicate indices or that they may contain duplicates (MATLAB allows for duplicates).

2.25. The MATLAB statement `A(i,j)=C`, where `i` and `j` are integer vectors, replaces the entries in the `A(i,j)` submatrix with the `length(i)`-by-`length(j)` matrix `C`. Write a function that performs this operation. Either assume that `i` and `j` do not contain duplicate indices or that they may contain duplicates (MATLAB allows for duplicates).

2.26. Write a function combining `cs_permute` and `cs_transpose` that computes the permuted transpose, just as in the MATLAB statement `C=A(p,q)'`, where `p` and `q` are permutation vectors. It should use one pass over the matrix to count the number of entries in `C` and another to copy entries from `A` to `C`.

2.27. Create three versions of `cs_gaxpy` that operate on dense matrices `X` and `Y` (`A` is still sparse). The first should assume `X` and `Y` are in column-major form. The second should use row-major form. The third should use column-major form but operate on blocks of (say) 32 columns of `X` at a time. Compare their performance.

2.28. Repeat Problem 2.27 but for `cs_gatxpy` instead (described in Problem 2.1).

2.29. Write four functions that modify a sparse matrix `A`, adding `k` *empty* rows or columns (an empty row or column has no entries in it). Adding empty rows takes $O(|A|)$ if added to the top or $O(1)$ if added to the bottom. Adding empty columns takes $O(n+k)$ time.

2.30. Experiment with the time taken by the MATLAB statement `r=A(i,:)` for an `m`-by-`n` matrix and a scalar `i`. Does MATLAB use a binary search (taking $O(\sum \log |\mathtt{A(:,j)}|)$ time)? Or does it use a linear search of each column? Does it exploit special cases, such as `r=A(1,:)` and `r=A(m,:)`?

2.31. Which CSparse functions work properly if duplicate entries are present?