Sparse Matrix Methods
Chapter 1-2 lecture notes

Tim Davis

2011

Chapter 1: Introduction

NOTE: these slides are terse. I only use them for material that I do not want to write on the board (mainly code, figures, and long equations). Most of the discussion, derivations, main points, basic equations, etc, will be on the chalk board. Thus, these slides are not meant as a stand-alone outline of the topic.

Block matrix

- rows of an m-by-n matrix A partitioned into r subsets
- columns are partitioned into c subsets
- A can be written as a block r-by-c matrix

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1c} \\ \vdots & & \vdots \\ A_{r1} & \cdots & A_{rc} \end{bmatrix}$$

- where A_{ij} is m_i-by-n_j, if m_i is the size of the ith row subset, and n_j is the size of the jth column subset.

Block matrix

- $C = AB$
- columns of A are partitioned identically to the rows of B.

$$C_{ij} = \sum_{k=1}^{c} A_{ik} B_{kj}$$

Chapter 2: Basic algorithms

CSparse matrix

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

Triplet form:

```
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 } ;
```

compressed column form:

```
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 } ;
```

CSparse data 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 ;
```

Matrix-vector multiplication

$z = Ax + y$, where y and x are dense vectors and A is sparse.

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

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

gaxpy

```
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))
```

malloc wrappers

```
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 */
}

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 */
}
```

allocate/free a sparse matrix

```
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 */
}
```

re-allocate a sparse matrix

```
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) ;
}
```

triplet form

Create:

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

Add one entry:

```
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) ;
}
```

convert triplet to compressed-column

```
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 */
}
```

wrap-up

```
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 */
}
```

cumulative sum

```
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]) */
}
```

transpose

```
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 */
}
```

```
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++)
    {
        q = nz ;                                    /* column j will start at q */
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            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 */
}
```

dropping entries from a matrix

```
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) ;
}
```

dropping entries

Dropping zeros:

```
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 */
}
```

Dropping small entries:

```
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 */
}
```

Matrix multiplication

- $C = AB$ should access A and B by column.
- let 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:

$$C_{*j} = \begin{bmatrix} A_{*1} & \cdots & A_{*k} \end{bmatrix} \begin{bmatrix} b_{1j} \\ \vdots \\ b_{kj} \end{bmatrix} = \sum_{i=1}^{k} A_{*i} b_{ij}.$$

Nonzero pattern of $C = AB$

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

Theorem (2.1)

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

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

```
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 ;
    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++)
        {
            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) ;
}
```

Matrix addition

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

Matrix addition

```
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 */
}
```

Vector permutation

`[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`

```
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) ;
}


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) ;
}
```

Invert a permutation vector

- permutation vector, `p[k]=i` if "old" row `i` is the k_{th} "new" row.
- inverse permutation vector, `pinv[i]=k`

```
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 */
}
```

Matrix permutation

$C = PAQ$, or `C=A(p,q)` in MATLAB.

```
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)) ;
}
```

Symmetric matrix permutation

```
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 */
}
```

matrix 1-norm

```
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) ;
}
```

reading matrix from a 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) ;
}
```

```
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) ;
}
```