Sparse Matrix Methods
Chapter 3 lecture notes

Tim Davis

2011

Chapter 2: Solving triangular systems

Dense right-hand side

- 2-by-2 block matrix:

$$\begin{bmatrix} l_{11} & 0 \\ l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

- L_{22} is $(n-1)$-by-$(n-1)$
- l_{21}, x_2, and b_2 are columns of length $n-1$
- l_{11}, x_1, and b_1 are scalars
- Leads to two equations:
  (1) $l_{11}x_1 = b_1$
  (2) $l_{21}x_1 + L_{22}x_2 = b_2$
- Solve (1), then recursively solve (2): $L_{22}x_2 = b_2 - l_{21}x_1$

Dense right-hand side

$x = b$
for $j = 0$ to $n - 1$ **do**
  $x_j = x_j / l_{jj}$
  for each $i > j$ for which $l_{ij} \neq 0$ **do**
    $x_i = x_i - l_{ij} x_j$

```
int cs_lsolve (const cs *L, double *x)
{
    int p, j, n, *Lp, *Li ;
    double *Lx ;
    if (!CS_CSC (L) || !x) return (0) ;                    /* check inputs */
    n = L->n ; Lp = L->p ; Li = L->i ; Lx = L->x ;
    for (j = 0 ; j < n ; j++)
    {
        x [j] /= Lx [Lp [j]] ;
        for (p = Lp [j]+1 ; p < Lp [j+1] ; p++)
        {
            x [Li [p]] -= Lx [p] * x [j] ;
        }
    }
    return (1) ;
}
```

Solving $L^T x = b$

$$\begin{bmatrix} l_{11} & l_{21}^T \\ 0 & L_{22}^T \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

```
int cs_ltsolve (const cs *L, double *x)
{
    int p, j, n, *Lp, *Li ;
    double *Lx ;
    if (!CS_CSC (L) || !x) return (0) ;                     /* check inputs */
    n = L->n ; Lp = L->p ; Li = L->i ; Lx = L->x ;
    for (j = n-1 ; j >= 0 ; j--)
    {
        for (p = Lp [j]+1 ; p < Lp [j+1] ; p++)
        {
            x [j] -= Lx [p] * x [Li [p]] ;
        }
        x [j] /= Lx [Lp [j]] ;
    }
    return (1) ;
}
```

Solving $Ux = b$

$$\begin{bmatrix} U_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

```
int cs_usolve (const cs *U, double *x)
{
    int p, j, n, *Up, *Ui ;
    double *Ux ;
    if (!CS_CSC (U) || !x) return (0) ;                    /* check inputs */
    n = U->n ; Up = U->p ; Ui = U->i ; Ux = U->x ;
    for (j = n-1 ; j >= 0 ; j--)
    {
        x [j] /= Ux [Up [j+1]-1] ;
        for (p = Up [j] ; p < Up [j+1]-1 ; p++)
        {
            x [Ui [p]] -= Ux [p] * x [j] ;
        }
    }
    return (1) ;
}
```

Solving $U^T x = b$

```
int cs_utsolve (const cs *U, double *x)
{
    int p, j, n, *Up, *Ui ;
    double *Ux ;
    if (!CS_CSC (U) || !x) return (0) ;                    /* check inputs */
    n = U->n ; Up = U->p ; Ui = U->i ; Ux = U->x ;
    for (j = 0 ; j < n ; j++)
    {
        for (p = Up [j] ; p < Up [j+1]-1 ; p++)
        {
            x [j] -= Ux [p] * x [Ui [p]] ;
        }
        x [j] /= Ux [Up [j+1]-1] ;
    }
    return (1) ;
}
```

$Lx = b$: sparse right-hand side

- Assume that L has a unit diagonal.

$x = b$
for $j = 0$ to $n-1$ **do**
  if $x_j \neq 0$
    for each $i > j$ for which $l_{ij} \neq 0$ **do**
      $x_i = x_i - l_{ij} x_j$

- Problem: time $O(n + |b| + f)$, where $f =$ flop count.

$Lx = b$: sparse right-hand side

- A better method: know the pattern of x before-hand.
- Let $\mathcal{X} = \{j \mid x_j \neq 0\}$.

$x = b$
for each $j \in \mathcal{X}$ **do**
  for each $i > j$ for which $l_{ij} \neq 0$ **do**
    $x_i = x_i - l_{ij} x_j$

- time $O(|b| + f)$, but how do we find $\mathcal{X}$?

$Lx = b$: finding $\mathcal{X}$

- $b_i \neq 0 \Rightarrow x_i \neq 0$
- $x_j \neq 0 \wedge \exists i (l_{ij} \neq 0) \Rightarrow x_i \neq 0$

Theorem (3.1)

Define the directed graph $G_L = (V, E)$ with nodes $V = \{1 \ldots n\}$ and edges $E = \{(j, i) \mid l_{ij} \neq 0\}$. Let $\mathrm{Reach}_L(i)$ denote the set of nodes reachable from node i via paths in G_L, and let $\mathrm{Reach}(\mathcal{B})$, for a set $\mathcal{B}$, be the set of all nodes reachable from any node in $\mathcal{B}$. The nonzero pattern $\mathcal{X} = \{j \mid x_j \neq 0\}$ of the solution x to the sparse linear system $Lx = b$ is given by $\mathcal{X} = \mathrm{Reach}_L(\mathcal{B})$, where $\mathcal{B} = \{i \mid b_i \neq 0\}$, assuming no numerical cancellation.

$Lx = b$: finding $\mathcal{X}$

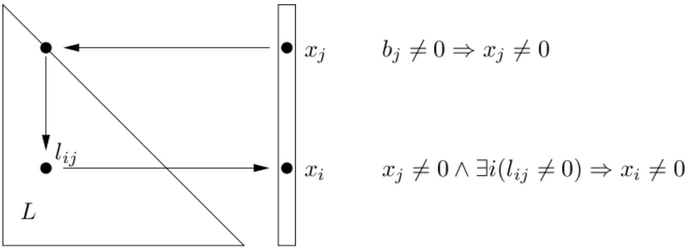

Figure 3.1. *Sparse triangular solve*

$Lx = b$: finding $\mathcal{X}$

```
function 𝒳 = reach (L, ℬ)
    assume all nodes are unmarked
    for each i for which b_i ≠ 0 do
        if node i is unmarked
            dfs (i)

function dfs (j)
    mark node j
    for each i for which l_ij ≠ 0 do
        if node i is unmarked
            dfs (i)
    push j onto stack for 𝒳
```

$Lx = b$: finding $\mathcal{X}$ recursively

```
int reachr (const cs *L, const cs *B, int *xi, int *w)
{
    int p, n = L->n ;
    int top = n ;                                 /* stack is empty */
    for (p = B->p [0] ; p < B->p [1] ; p++)      /* for each i in pattern of b */
    {
        if (w [B->i [p]] != 1)                    /* if i is unmarked */
        {
            dfsr (B->i [p], L, &top, xi, w) ;     /* start a dfs at i */
        }
    }
    return (top) ;                                /* return top of stack */
}

void dfsr (int j, const cs *L, int *top, int *xi, int *w)
{
    int p ;
    w [j] = 1 ;                                   /* mark node j */
    for (p = L->p [j] ; p < L->p [j+1] ; p++)    /* for each i in L(:,j) */
    {
        if (w [L->i [p]] != 1)                    /* if i is unmarked */
        {
            dfsr (L->i [p], L, top, xi, w) ;      /* start a dfs at i */
        }
    }
    xi [--(*top)] = j ;                           /* push j onto the stack */
}
```

$Lx = b$: finding $\mathcal{X}$ non-recursively

```
#define CS_FLIP(i) (-(i)-2)
#define CS_UNFLIP(i) (((i) < 0) ? CS_FLIP(i) : (i))
#define CS_MARKED(w,j) (w [j] < 0)
#define CS_MARK(w,j) { w [j] = CS_FLIP (w [j]) ; }

int cs_reach (cs *G, const cs *B, int k, int *xi, const int *pinv)
{
    int p, n, top, *Bp, *Bi, *Gp ;
    if (!CS_CSC (G) || !CS_CSC (B) || !xi) return (-1) ;    /* check inputs */
    n = G->n ; Bp = B->p ; Bi = B->i ; Gp = G->p ;
    top = n ;
    for (p = Bp [k] ; p < Bp [k+1] ; p++)
    {
        if (!CS_MARKED (Gp, Bi [p]))    /* start a dfs at unmarked node i */
        {
            top = cs_dfs (Bi [p], G, top, xi, xi+n, pinv) ;
        }
    }
    for (p = top ; p < n ; p++) CS_MARK (Gp, xi [p]) ;  /* restore G */
    return (top) ;
}
```

```
int cs_dfs (int j, cs *G, int top, int *xi, int *pstack, const int *pinv)
{
    int i, p, p2, done, jnew, head = 0, *Gp, *Gi ;
    if (!CS_CSC (G) || !xi || !pstack) return (-1) ;    /* check inputs */
    Gp = G->p ; Gi = G->i ;
    xi [0] = j ;                  /* initialize the recursion stack */
    while (head >= 0)
    {
        j = xi [head] ;           /* get j from the top of the recursion stack */
        jnew = pinv ? (pinv [j]) : j ;
        if (!CS_MARKED (Gp, j))
        {
            CS_MARK (Gp, j) ;         /* mark node j as visited */
            pstack [head] = (jnew < 0) ? 0 : CS_UNFLIP (Gp [jnew]) ;
        }
        done = 1 ;                    /* node j done if no unvisited neighbors */
        p2 = (jnew < 0) ? 0 : CS_UNFLIP (Gp [jnew+1]) ;
        for (p = pstack [head] ; p < p2 ; p++)  /* examine all neighbors of j */
        {
            i = Gi [p] ;              /* consider neighbor node i */
            if (CS_MARKED (Gp, i)) continue ;   /* skip visited node i */
            pstack [head] = p ;       /* pause depth-first search of node j */
            xi [++head] = i ;         /* start dfs at node i */
            done = 0 ;                /* node j is not done */
            break ;                   /* break, to start dfs (i) */
        }
        if (done)                 /* depth-first search at node j is done */
        {
            head-- ;              /* remove j from the recursion stack */
            xi [--top] = j ;      /* and place in the output stack */
        }
    }
    return (top) ;
}
```

```
int cs_spsolve (cs *G, const cs *B, int k, int *xi, double *x, const int *pinv,
    int lo)
{
    int j, J, p, q, px, top, n, *Gp, *Gi, *Bp, *Bi ;
    double *Gx, *Bx ;
    if (!CS_CSC (G) || !CS_CSC (B) || !xi || !x) return (-1) ;
    Gp = G->p ; Gi = G->i ; Gx = G->x ; n = G->n ;
    Bp = B->p ; Bi = B->i ; Bx = B->x ;
    top = cs_reach (G, B, k, xi, pinv) ;         /* xi[top..n-1]=Reach(B(:,k)) */
    for (p = top ; p < n ; p++) x [xi [p]] = 0 ;    /* clear x */
    for (p = Bp [k] ; p < Bp [k+1] ; p++) x [Bi [p]] = Bx [p] ; /* scatter B */
    for (px = top ; px < n ; px++)
    {
        j = xi [px] ;                               /* x(j) is nonzero */
        J = pinv ? (pinv [j]) : j ;                 /* j maps to col J of G */
        if (J < 0) continue ;                       /* column J is empty */
        x [j] /= Gx [lo ? (Gp [J]) : (Gp [J+1]-1)] ;/* x(j) /= G(j,j) */
        p = lo ? (Gp [J]+1) : (Gp [J]) ;            /* lo: L(j,j) 1st entry */
        q = lo ? (Gp [J+1]) : (Gp [J+1]-1) ;        /* up: U(j,j) last entry */
        for ( ; p < q ; p++)
        {
            x [Gi [p]] -= Gx [p] * x [j] ;          /* x(i) -= G(i,j) * x(j) */
        }
    }
    return (top) ;                                  /* return top of stack */
}
```

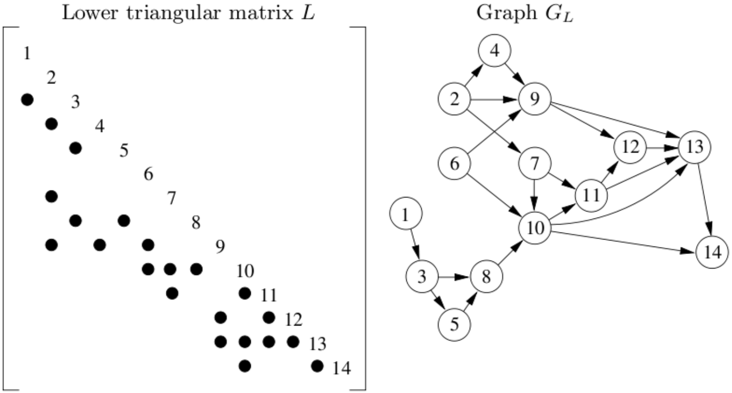

Figure 3.2. *Solving $Lx = b$ where L, x, and b are sparse*