C 언어의 유한요소해석 프로그램을 위한 Skyline Algorithm

Skyline Algorithm for Finite Element Analysis Programs Written in C Language

이 재 영*
Lee, Jae Young

요 약

이 논문은 C 언어의 특성에 맞도록 skyline algorithm을 수정하여 제시하였다. 수정된 algorithm은 FORTRAN을 위한 종래의 algorithm에 비해서 프로그램의 구조를 개선하고 효율성을 높혀주는 이점이 있다. 강성행열의 조립과 분해를 단순화시키므로 프로그램의 실행시간이 현저히 단축된다. 장차 유한요소해석 프로그램의 개발에 실용적으로 활용될 수 있도록 C로 작성한 skyline algorithm의 원시프로그램 리스트를 수록하였다.

Abstract

A modified skyline algorithm suitable for C language is proposed in this paper. The modified algorithm improves the computational efficiency and the structure of the program. Substantial reduction of execution time is achieved by simplifying assemblage and decomposition of the stiffness matrix. A source program is also provided for use in future development of finite element softwares.

서 론

지금까지 개발된 유한요소 해석 프로그램들은 거의 모두 FORTRAN으로 작성되었다. FORTRAN은 지난 1950년대말 이후에 과학기술용 프로그래밍 언어로서 지배적인 위치를 차지해 왔다. 그러나, 근년에는 이 언어의 비효율성이 자주 지적되어 왔으며, 그 활용도가 감소하는 추세에 있다.[7] FORTRAN을 대체해서 많이 활용되고 있는 언어 중의 하나는 C언어이다. 이 언어는 자료구조, 효율성, modularity 및 portability등에 있어서 다른 언어보다 뛰어나며, Assembly 언어와 같은 low level의 프로그래밍이 가능하므로 향후의 유한요소해석 프로그램개발에 많이 채택될 것으로 전망된다.

유한요소해석에서 방정식을 조립하고 풀어내는 과정은 많은 계산기간을 점유하는 부분이다. 방정

* 정회원, 전북대학교 농공학과 조교수

이 논문에 대한 토론을 1989년 9월30일까지 본학회에 보내주시면 1990년 3월호에 그 결과를 게재하겠습니다.

식의 조립해법은 in- core solution과 out-of-core solution 으로 대별된다. 컴퓨터의 random access memory의 단가가 저렴해지고, 용량이 대형화됨에 따라서, 점차 in - core solution이 유리해질 것이다.[3] Skyline algorithm은 가장 많이 이용되어온 in-core solution 방법이고,[26] 이를 위한 FORTRAN 프로그램은 교과서적인 내용이 되었으며[1], 실제로 많은 유한요소해석프로그램의 부프로그램으로 이용되어왔다.[4]

아직까지 C언어를 위하여 특별히 고안된 solution algorithm이 문헌에 나타난 바는 없다. 그러므로 새로운 유한요소해석 프로그램을 C 언어로 개발할 경우에는 상술한 바와 같은 기존의 FORTRAN 프로그램을 이 언어로 번역하여 삽입하는 것이 가장 손쉬운 방법일 것이다. 그러나 프로그램의 효율성을 높히기 위해서는 C 언어에 의한 프로그램은 이와같은 FORTRAN 프로그램의 단순한 번역이 아니라, C 언어의 특성에 맞고, 또 이 언어의 장점을 최대한으로 이용한 새로운 algorithm에 의한것이 바람직하다. 이 논문은 이러한 관점에서 기존의 skyline algorithm을 수정하여 개선된 algorithm 을 제시하였다. 수정된 algorithm은 C 언어의 장점인 pointer variable과 dynamic dimensioning을 적극적으로 활용하여 skyline mapping의 과정을 거치지 않고 방정식을 간단하게 조립하도록 하였으며, 방정식의 분해과정에서 테스트와 분기를 가급적 피하고, book keeping을 배제하므로서 프로그램의 구조를 개선하고 계산시간을 상당히 단축할 수 있도록 하였다.

Skyline Algorithm의 원리

Skyline algorithm은 profile algorithm이라고도 하며, sparse symmetric matrix를 효율적으로 풀기위해서 Gauss소거법을 변형시킨 방법이다. 이 방법의 핵심은 강성행렬을 LDL^T의 형태로 삼각행렬 L과 대각선행렬 D로 분해하여 행렬의 유효부분(값이 零이 아닌 부분) 내에서만 계산이 실행되도록 하므로써 memory의 요소를 최소화하는 것이다. 강성행렬, 변위벡터 및 하중벡터를 각각 K, △ 및 F라고 하면 평형방정식은

$$K\triangle = F$$

인바, 대칭인 강성행렬을

$$K = LDL^T$$

로 분해하면, 이와 동시에 방정식 우변의 하중벡터는

$$F = LV$$

로 분해되어 새로운 벡터V를 얻게 된다. 따라서

$$DL^T = V$$

의 관계가 성립되므로 역대입과정을 통해서

$$L^T\triangle = D^{-1}V$$

으로부터 △가 산정된다. LDL^T 분해에 관해서는 참고문헌1에 구체적으로 설명되어 있으며, 여기서는 이를 생략한다.

Skyline Algorithm의 개선

Skyline algorithm은 band matrix algorithm보다 memory의 소요를 줄일수 있으며,절점번호에 따른 소요 memory의 증감이 후자보다 덜 민감하다는 장점이 있다.[5] 그러나 종래의 skyline algorithm에 의하면 그림1(a)와 같이 skyline map을 만들고, 이에 의거하여 요소강성행렬을 일차원 배열로 조립하기 때문에, 번거로운 book keeping이 따라야하고, 조립 방법이 까다롭다. 이는 FORTRAN에 의한 momory 관리나 변수의 addressing 방식이 지니고 있는 한계성으로 인하여 불가피하다. 그러므로 C와 같이 dynamic memory allocation이 가능하고, pointer variable에 의한 집합적 addressing이 가능한 프로그래밍 언어를 사용할 경우에 FORTRAN 방식의 algorithm을 그대로 적용하는 것은 불합리하다.

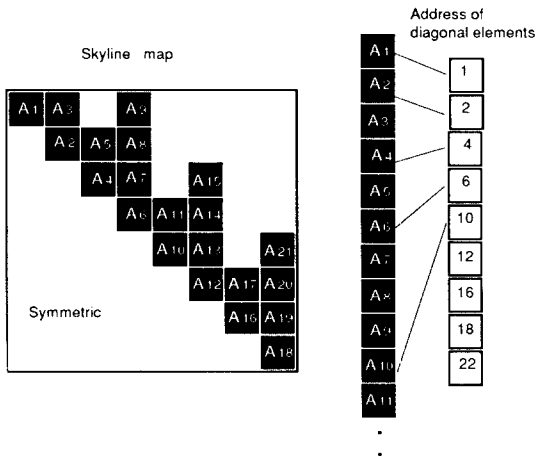
그림1에서 검은색으로 표시한 행렬요소는 값이 零이 아닌 유효부분, 즉 skyline 안에 있는 부분이며, 흰색으로 표시하거나 빈공간으로 표시한 요소는 값이 零인 부분, 즉 skyline 밖에 있는 부분이다. skyline algorithm에 의하면 계산이 강성행렬의 skyline 안에서만 이루어 지기 때문에 이 부분에만 기억장소를 할당하는 것이 바람직하다. 이를 위해서 FORTRAN 프로그램에서는 그림1(a)와 같이 skyline 안쪽만을 따서 일차원 array로 정렬한다. 또한 이와같이 일차원 array를 이용하여 variable

dimensioning을 하므로서 memory의 낭비를 배제할 수 있다. 그러나 전체강성행렬을 조립하기 위해서는 먼저 강성행렬의 대각선 위치를 지정하는 skyline map을 만들어야하고 요소강성행렬이 일차원 array에서 차지하는 위치를 계산하여야 한다. 따라서 강성행렬의 조립과정이 복잡해질 수 밖에 없다. 그뿐 아니라 LDL^T 분해가 일차원 array 상에서 실행되므로 계산이 비효율적이다.

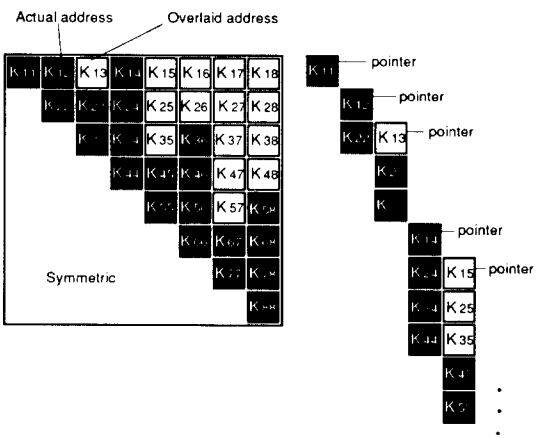
이 논문에서 제시한 algorithm에 의하면 일차원 array와 동일한 크기의 memory를 차지하면서 강성행렬을 정방행렬의 형태로 조립하므로서 FORTRAN 프로그램에서와 같은 복잡성과 계산상의 비효율성을 피할수 있다. 이는 C 언어 특유의 pointer array에 의해 memory를 동적으로 할당하고, 그림1(b)와 같이 강성행렬의 skyline 內外의 부분이 기억장소를 공유하도록 하므로서 달성된다. 그 순서는 다음과 같다.

1. 전체 강성행렬의 각 列의 skyline 길이와 전체길이를 계산한다.
2. Skyline의 전체길이 만큼 memory를 할당한다.
3. 강성행렬의 각 列이 차지할 memory의 위치를 계산한다.
4. Ponter array를 이용하여 강성행렬의 각 列을 앞에서 할당한 memory에 배치한다.

Pointer array의 i요소는 강성행렬 i번째 列의 address 를 지시한다. 그림2의 flow chart에 나타낸 바와 같이 이 address는 나중에 skyline의 column height와 경계를 계산하는데 이용된다. 이때 그림1(b)와 같이 실질적으로는 강성행렬의 skyline이 일렬로 밀집되도록 하면서, 관념상으로는 강성행렬이 정방행렬의 형태를 유지하도록 1行의 위치를 정해준다. 일단 이와 같이 강성행렬의 memory 위치가 정해지면, skyline의 길이나, 행렬 대각선의 위치등에 대해서는 더이상 상관할 필요가 없으며, 강성행렬을 원래의 정방행렬과 같이 다룰 수 있다. 따라서 강성행렬의 조립과 LDL^T 분해가 FORTRAN을 위한 원래의 algorithm 보다 훨씬 간단해진다.



(a) FORTRAN을 위한 algorithm



(b) C에 맞도록 수정된 algorithm

그림1. Skyline Algorithm에 의한 강성행렬의 저장

Algorithm의 효율성 검토

수정된 algorithm의 이점은 실제의 프로그램 리스트와 그 실행 속도를 비교해보면 쉽게 알 수 있다. 부록1과 부록2는 각각 수정된 algorithm과 종래의 algorithm에 따라서 강성행렬을 LDT^T 분해하여 풀어내는 프로그램들이다. 후자는 원래 FORTRAN 으로 작성된 것을 C 언어로 번역한 것이다. 표1에서 보는 바와 같이 프로그램의 길이

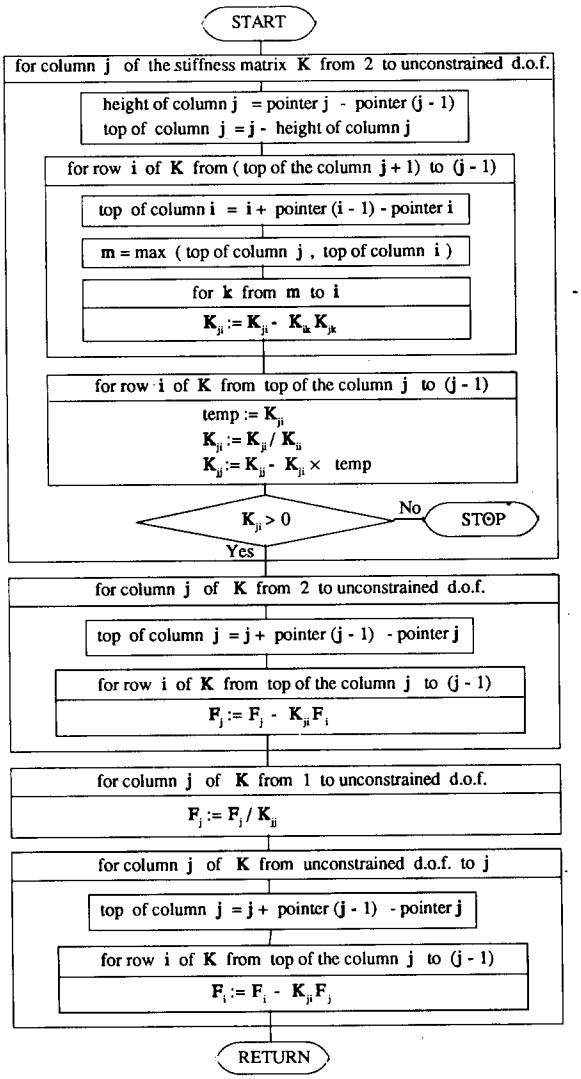


그림2. 수정된 skyline algorithm의 flow chart

를 비교하면 거의 두배의 차이가 있다. 또한 FORTRAN 식의 algorithm에 의하면 테스트와 분기를 거치므로 프로그램의 흐름이 복잡해질 수 밖에 없으나, 수정된 algorithm에 따르면 이러한 테스트와 분기를 피할 수 있기 때문에 그림2의 flow chart에서 볼 수 있는 바와 같이 프로그램의 구조가 간결해지며, 체계적으로 이루어질 수 있음을 알수 있다.

수정된 algorithm의 효율성을 입증하기 위하여 표2와 같은 4개의 테스트 문제를 설정하고, 이들

표1. 프로그램 길이의 비교

		No. of statements	No. of tests	No. divergences
Present algorithm (Appendix 1)	Assemblage	16	2	0
	Decomposition	31	2	1
Old algorithm (Appendix 2)	Assemblage	22	3	0
	Decomposition	63	7	6

표2. 테스트 문제의 크기

Test Problem No.		I	II	III	IV
Size of problems	No of elements	8	32	72	128
	Unconstrained d.o.f.	96	352	768	1344
	Total skyline length	2671	20395	67559	158563
	Critical front length	60	80	100	120

표3. 실행시간*의 비교(단위 : 秒)

Test Problem No.		I	II	III	IV
Present algorithm (Modified skyline solver)	Assemblage	0.16	0.82	2.03	3.68
	Decomposition	1.60	21.31	79.97	188.49
	Total	1.76	22.13	82.00	192.17
Old algorithm (Skyline solver translated from FORTRAN into C)	Assemblage	0.17	0.93	2.19	4.07
	Decomposition	2.25	32.13	124.38	299.36
	Total	2.42	33.06	126.57	303.43
Frontal solver written in C	Assemblage and decomposition	4.71	44.95	158.26	361.08

* Processor: I 80386(20MHz), Coprocessor : I 80387 (20MHz), 1 wait state에서 실행

의 실행시간을 동일한 조건에서 측정하였다. 표3은 수정된 algorithm과 재래적인 algorithm의 skyline algorithm와 C 언어로 번역된 frontal solver에 의한 실행시간을 비교한 것이다. Frontal solution algorithm이 skyline algorithm 보다 실행속도에 있어서 불리하다는 것은 이미 잘 알려진 바와 같다. 이는 C 언어로 프로그램을 작성했을 경우에도 마찬가지임을 알수 있다. 표3에 나타난 중요한 결과는 수정된 algorithm에 의하여 skyline algorithm의 실행시간이 현저히 단축된다는 점이다. 실행시간의 비율은 전체 자유도가 클수록 다소 커져서 약 1 : 1.6까지 달한다.

강성행렬의 조립에 소요되는 시간(요소강성행렬의 계산시간을 제외)도 표3에 나타난 바와 같이 수정된 algorithm에 의하여 약 10%정도 단축될 수 있으나, 유한요소해석 전체의 소요시간에서 차지하는 비중이 작기 때문에 그 자체가 중요한 의

미를 갖지는 않는다. 그러나 이는 강성행렬 조립 과정의 단순화를 반영하는 것이며, 이로인하여 coding 이 용이해진다는 것은 커다란 이점이다. 이러한 이점은 특히 비선형해석등에 있어서 강성행렬의 일부분만을 update할 필요가 있을 경우나 또는 Eigen value해석 등의 경우에 더욱 두드러질 것으로 판단된다.

결 론

C로 작성되는 유한요소해석 프로그램에 FORTRAN 방식의 algorithm을 적용하는 것은 불합리하다. 이 논문은 C 언어의 특성에 맞도록 수정된 skyline algorthm을 제시하였다. 이 algorithm은 프로그램의 구조를 개선하고 계산상의 효율성을 높혀준다. 특히 강성행렬을 원래의 정방 행렬과 같이 다룰 수 있으므로 skyline algorithm 과정이 불필요하고, 조립이 간단해지며, LDL^T 분해에 수반되는 많은 테스트와 분기를 배제하므로서 실행속도를 1.6배 이상높힐 수 있다. 이 논문에 수록된 skyline algorithm의 프로그램은 장차 C 언어로 작성되는 유한요소해석 프로그램의 한 부분으로 유용하게 쓰일 것으로 기대된다.

참 고 문 헌

1. Bathe, K－J., and E.L. Wilson, *Numerical Methods in Finite Element Analysis*, Prentice Hall, Inc., New Jersey, 1976.
2. Elwi, A.E., and D.W. Murray, Skyline algorithms for multilevel substructure analysis, *International Journal for Numerical Methods in Engineering*, Vol. 21, 1985, pp.465－479.
3. Jalon, J.G., and R. Muguerza, On the mini－computer solution of large systems of linear equations arising from the finite element method, *Applied Mathematical Modelling*, Vol. 40, 1980, pp.381－388.
4. Nathan I., and L. William, Solution of linear equations for small computer systems, *International Journal for Numerical Methods in Engineering*, Vol. 20, 1984, pp.625－641.
5. Sloan, S.W., An algorithm for profile and wavefront reduction of sparse metrices, *International Journal for Numerical Methods in Engineering*, Vol. 23, 1986, pp.239－251.
6. Taylor, R.L., E.L. Wilson, and S.J. Sackette, Direct solution of equations by frontal and variable band active column methods, *Nonlinear Finite Element Analysis Structural Mechanics*, Springer Verlag, 1981.
7. Tesler, L.G., Programming languages, *Scientific American*, Sept., 1984., pp.58－66.

(접수일자 1989. 4. 1)

부록1 수정된 algorithm에 의한 LDL^T decomposition 프로그램

이 프로그램에 사용된 external variable은 다음과 같다.

gsm : global stiffness matrix
gfv : global force vector
free_dof : unconstrained d.o.f.
total_dof : total d.o.f.
nodal_dof : nadal d.o.f.
ordered_dof : sequencial order of unconstrained d.o.f.

이 프로그램의 실행에 앞서서 다음과 같은 memory allocation이 필요하다.

```
arrays=(double *)calloc(total_ent,sizeof(double));
for(i=0;i<free_dof;i++) {
        head=sky_pivot[i+1]-i-1;
        gsm[i]=arrays+head;
}
```

```
ldlt_decompose()
{
        int i,j,k,id,mi,mj,mm,nn;
        double c;
        for(j=1;j<free_dof;j++) {
                mj=j+gsm[j-1]-gsm[j];
                        for(i=mj+1;i<j;i++) {
                                mm=(mj>(mi=i+gsm[i-1]-gsm[i])) ? mj : mi;
                                for(k=mm;k<i;k++)
                                        gsm[j][i]-=gsm[i][k]*gsm[j][k];
                }
                for(i=mj;i<j;i++) {
                        c=gsm[j][i];
                        gsm[j][i]/=gsm[i][i];
                        gsm[j][j]-=c*gsm[j][i];
                }
                if(gsm[j][j]<=0.0) {
                        printf("\n Unstable system !");
                        return(0);
                }
        }
        for(j=1;j<free_dof;j++) {
                mj=j+gsm[j-1]-gsm[j];
                for(i=mj;i<j;i++)
                        gfv[j]-=gsm[j][i]*gfv[i];
        }
        for(j=0;j<free_dof;j++)
                gfv[j]/=gsm[j][j];
        for(j=free_dof-1;j>0;j--) {
                        mj=j+gsm[j-1]-gsm[j];
                 for(i=mj;i<j;i++)
                        gfv[i]-=gsm[j][i]*gfv[j];
        }
        for(nn=total_dof-1;nn>=0;nn--) {
                i=nn/nodal_dof;
                k=nn-i*nodal_dof;
                id=ordered_dof[k][i];
                if(id>=0)
                        gfv[nn]=gfv[id];
                else
                        gfv[nn]=0;
        }
        return(1);
}
```

부록 2　FORTRAN에서 C로 번역한 LDL^T decomposition 프로그램

이 프로그램에 사용된 external variable은 다음과 같다.

gsm : global stiffness matrix
gfv : global force vector
free_dof : unconstrained d.o.f.
total_dof : total d.o.f.
nodal_dof : nadal d.o.f.
ordered_dof : sequencial order of unconstrained d.o.f.

이 프로그램의 실행에 앞서서 다음과 같은 memory allocation이 필요하다.

```
gfv=(double *)calloc(total_dof,sizeof(double));
gsm=(double *)calloc(total_ent,sizeof(double));
```

```
ldlt_decompose()
{
        int i,k,n,l,id,nc,nd,nh,ni,nj,nk,nl,nll,nlt,nn,nu;
        double b,c;
        for(i=0;i<free_dof;i++) {
                nk=sky_pivot[i];
                nl=nk+1;
                nu=sky_pivot[i+1]-1;
                nh=nu-nl;
                if(nh>0) {
                        n=i-nh;
                        nc=0;
                        nlt=nu;
                        for(nj=0;nj<nh;nj++) {
                                nc++;
                                nlt--;
                                ni=sky_pivot[n];
                                nd=sky_pivot[n+1]-ni-1;
                                if(nd>0) {
                                        nn = (nc<=nd) ? nc : nd;
                                        c=0;
                                        for(nll=1;nll<=nn;nll++) {
                                                c+=gsm[ni+nll]*gsm[nlt+nll];
                                        }
                                        gsm[nlt]-=c;
                                }
                                n++;
                        }
                }
                if(nh>=0) {
                        n=i;
                        b=0;
```

```
                for(nn=nl;nn<=nu;nn++) {
                        ni=sky_pivot[--n];
                        c=gsm[nn]/gsm[ni];
                        b+=c*gsm[nn];
                        gsm[nn]=c;
                }
                gsm[nk]-=b;
        }
        if(gsm[nk]<=0.0) {
                printf("\n Unstable system");
                return(0);
        }
}
        for(k=0;k<free_dof;k++) {
                nl=sky_pivot[k]+1;
                nu=sky_pivot[k+1]-1;
                if(nu-nl>=0) {
                        n=k;
                        c=0;
                        for(nn=nl;nn<=nu;nn++)
                                c+=gsm[nn]*gfv[--n];
                        gfv[k]-=c;
                }
        }
        for(k=0;k<free_dof;k++) {
                n=sky_pivot[k];
                gfv[k]/=gsm[n];
        }
        k=free_dof-1;
        for(l=1;l<free_dof;l++) {
                nl=sky_pivot[k]+1;
                nu=sky_pivot[k+1]-1;
                if((nu-nl)>=0) {
                        n=k;
                        for(nn=nl;nn<=nu;nn++) {
                                n--;
                                gfv[n]-=gsm[nn]*gfv[k];
                        }
                }
                k--;
        }
        for(nn=total_dof-1;nn>=0;nn--) {
                i=nn/nodal_dof;
                k=nn-i*nodal_dof;
                id=ordered_dof[k][i];
                if(id>=0)
                        gfv[nn]=gfv[id];
                else
                        gfv[nn]=0;
        }
        return(1);
}
```