```
/*****************************************************************************
*
*                          TWO DIMENSIONAL ELASTIC MEMBRANE
*
*
*                                    C++ Code
*
*                                Version: January 2009
*       Author: Arash Zamani, PhD from AmirKabir University of Technology
*
*
*             This code has been tested by Microsoft Visual C++ 2008
*****************************************************************************
*
*                         Introducing global variables
*
* nx=number of divisions in x direction
* ny=number of divisions in y direction
* nr=number of divisions in radial direction for an elliptic domain
* nc=number of divisions in angular direction for the first layer of
*      elements around the ellipse center
* npe=number of nodes per element
* nb=bandwith
* ne=number of elements
* nn=number of nodes
* node=connectivity
* xv=vector of x values
* yv=vector of y values
* iconv=convection vector
* const1=determines whether to impose primary boundary conditions
* const2=determines the value of primary boundary conditions
* maxh=height of each column for skyline implementation
* gk=global stiffness matrix
* p=the vector which is first used to store external forces and then
*   to store the calculated primary variables from finite element model
* Q=heat generation inside element (or pressure in membrane formulation).
* q=heat flux on the boundary edges
* h=the coefficient of convection heat transfer
* Tinf=temperature at infinity (surrounding media).
* kx,ky=thermal conductivities in x and y directions respectively
*       (or tension in membrane formulation).
*
*****************************************************************************/


#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <io.h>
#include <iostream>
#include <fstream>
#include <float.h>
using namespace std;

const long   nx=20, ny=32, nr=20, nc=6;


//the variables for a rectangular domain and three node triangular element
//(rd_3nte).
const long   npe=3, nb=nx+2, ne=2*nx*ny, nn=(nx+1)*(ny+1);

//the variables for a rectangular domain and six node triangular element
//(rd_6nte).
//const long   npe=6, nb=4*nx+3, ne=2*nx*ny, nn=(2*nx+1)*(2*ny+1);

//the variables for a triangular domain and three node triangular element
//(td_3nte).
//const long   npe=3, nb=ny+2, ne=ny*ny, nn=(ny+1)*(ny+2)/2;

//the variables for a triangular domain and six node triangular element
//(td_6nte).
```

```
//const long   npe=6, nb=4*ny+2, ne=ny*ny, nn=(2*ny+1)*(ny+1);

//the variables for an elliptic domain and three node triangular element
//(ed_3nte).
//const long   npe=3, nb=nc*(2*nr-3), ne=nc*(nr-1)*(nr-1), nn=nc*nr*(nr-1)/2+1;


long       node[npe*ne+1], iconv[nn+1], const1[nn+1],maxh[nn+2];
double     gk[nn*nb+1], p[nn+1], xv[nn+1], yv[nn+1], const2[nn+1];
double     Q[ne+1], h[nn+1], q[nn+1], Tinf[nn+1];
double     kx,ky;

void mesh_rd_3nte(double x0,double y0);
void mesh_rd_6nte(double x0,double y0);
void mesh_td_3nte(double x0,double y0);
void mesh_td_6nte(double x0,double y0);
void mesh_ed_3nte(double x0,double y0);
void bandwidth();
void core_3nte(char type);
void core_6nte(char type);
void boundary_sym_banded(long neqns,long nbw,double mat[],double rhs[],
                         long cond1[],double cond2[]);
void boundary_sym_skyline(long neqns,long nbw,double mat[],double rhs[],
                          long maxh[],long cond1[],double cond2[]);
void solve_gauss_sym_banded(long neqns,long nbw,double mat[],double rhs[]);
void solve_LUdecom_sym_skyline(long neqns,double mat[],double rhs[],long maxh[]);
void file_out();


void main(void)
{
    char type='b'; //specifies whether to choose the banded or skyline form.
                   //the value 'b' refers to banded form.
                   //the value 's' refers to skyline form.

    mesh_rd_3nte(5.0,8.0);

    switch(type){
        case 'b' :
            core_3nte(type);
            boundary_sym_banded(nn,nb,gk,p,const1,const2);
            solve_gauss_sym_banded(nn,nb,gk,p);
            break;
        case 's' :
            bandwidth();
            core_3nte(type);
            boundary_sym_skyline(nn,nb,gk,p,maxh,const1,const2);
            solve_LUdecom_sym_skyline(nn,gk,p,maxh);
            break;
    }

    file_out();
}


void mesh_rd_3nte(double x0,double y0)//generates the mesh for a rectangular
                                      //domain by three node triangular elements
{
    long i,j,n,m,n1,n2,nx1,ny1;

    nx1=nx+1;
    ny1=ny+1;
    m=-6;
    for (j=1;j<=ny;j++){
        for (i=1;i<=nx;i++){
            m+=6;
            n=nx1*(j-1)+i;

            node[m+1]=n;
            node[m+2]=n+1;
            node[m+3]=n+nx+1;

            node[m+4]=n+1;
```

```
            node[m+5]=n+nx+2;
            node[m+6]=n+nx+1;
        }
    }

    for (i=1;i<=nx1;i++){
        for (j=1;j<=ny1;j++){
            n=nx1*(j-1)+i;
            xv[n]=double(i-1)*x0/double(nx);
            yv[n]=double(j-1)*y0/double(ny);
        }
    }

    for (i=1;i<=nn;i++){
        iconv[i]=0;
        const1[i]=0;
        const2[i]=0.0;
    }

    n=nx1*ny;
    for (i=1;i<=nx1;i++){
        n1=i;
        n2=i+n;
        const1[n1]=1;
        const1[n2]=1;

        const2[n1]=0.0;
        const2[n2]=0.0;
    }

    for (j=2;j<=ny;j++){
        n1=nx1*(j-1)+1;
        n2=nx1*j;
        const1[n1]=1;
        const1[n2]=1;

        const2[n1]=0.0;
        const2[n2]=0.0;
    }

    for(i=1;i<=ne;i++){
        Q[i]=10.0;
    }

    kx=1.0;ky=1.0;
}


void mesh_rd_6nte(double x0,double y0)//generates the mesh for a rectangular
                                      //domain by six node triangular elements
{
    long i,j,n,m,n1,n2,nx1,ny1,ny2;

    nx1=2*nx+1;
    ny1=2*ny+1;
    m=-12;
    for (j=1;j<=ny;j++){
        for (i=1;i<=nx;i++){
            m+=12;
            n=2*nx1*(j-1)+2*i-1;

            node[m+1]=n;
            node[m+2]=n+2;
            node[m+3]=n+4*nx+2;
            node[m+4]=n+1;
            node[m+5]=n+2*nx+2;
            node[m+6]=n+2*nx+1;

            node[m+7]=n+2;
            node[m+8]=n+4*nx+4;
            node[m+9]=n+4*nx+2;
            node[m+10]=n+2*nx+3;
            node[m+11]=n+4*nx+3;
```

```
            node[m+12]=n+2*nx+2;
        }
    }

    for (i=1;i<=nx1;i++){
        for (j=1;j<=ny1;j++){
            n=nx1*(j-1)+i;
            xv[n]=double(i-1)*x0/double(2*nx);
            yv[n]=double(j-1)*y0/double(2*ny);
        }
    }

    for (i=1;i<=nn;i++){
        iconv[i]=0;
        const1[i]=0;
        const2[i]=0.0;
    }

    n=2*nx1*ny;
    for (i=1;i<=nx1;i++){
        n1=i;
        n2=i+n;
        const1[n1]=1;
        const1[n2]=1;

        const2[n1]=0.0;
        const2[n2]=0.0;
    }

    ny2=ny1-1;
    for (j=2;j<=ny2;j++){
        n1=nx1*(j-1)+1;
        n2=nx1*j;
        const1[n1]=1;
        const1[n2]=1;

        const2[n1]=0.0;
        const2[n2]=0.0;
    }

    for(i=1;i<=ne;i++){
        Q[i]=10.0;
    }

    kx=1.0;ky=1.0;
}


void mesh_td_3nte(double x0,double y0)//generates the mesh for a triangular
                                      //domain by three node triangular elements
{
    long i,j,n,m,n1,n2,n3,ny1;

    ny1=ny+1;
    for (j=1;j<=ny;j++){
        for (i=1;i<=j-1;i++){
            m=3*((j-1)*(j-1)+2*(i-1));
            n=(j-1)*j/2+i;

            node[m+1]=n;
            node[m+2]=n+j+1;
            node[m+3]=n+j;

            node[m+4]=n;
            node[m+5]=n+1;
            node[m+6]=n+j+1;
        }
    }

    for(j=1;j<=ny;j++){
            n=j*(j+1)/2;
            m=j*j;
```

```
            node[3*(m-1)+1]=n;
            node[3*(m-1)+2]=n+j+1;
            node[3*(m-1)+3]=n+j;
    }


    for (j=1;j<=ny1;j++){
        for (i=1;i<=j;i++){
            n=(j-1)*j/2+i;
            xv[n]=double(i-1)*x0/double(ny);
            yv[n]=double(j-1)*y0/double(ny);
        }
    }

    for (i=1;i<=nn;i++){
        iconv[i]=0;
        const1[i]=0;
        const2[i]=0.0;
    }

    n=ny*ny1/2;
    for (j=1;j<=ny1;j++){
        n1=j*(j-1)/2+1;
        n2=j+n;
        n3=j*(j+1)/2;

        const1[n1]=1;
        const1[n2]=1;
        const1[n3]=1;

        const2[n1]=0.0;
        const2[n2]=0.0;
        const2[n3]=0.0;
    }

    for(i=1;i<=ne;i++){
        Q[i]=10.0;
    }

    kx=1.0;ky=1.0;
}


void mesh_td_6nte(double x0,double y0)//generates the mesh for a triangular
                                      //domain by six node triangular elements
{
    long i,j,n,m,n1,n2,n3,ny1;

    ny1=2*ny+1;
    for (j=1;j<=ny;j++){
        for (i=1;i<=j-1;i++){
            m=6*((j-1)*(j-1)+2*(i-1));
            n=(j-1)*(2*j-1)+2*i-1;

            node[m+1]=n;
            node[m+2]=n+4*j+1;
            node[m+3]=n+4*j-1;
            node[m+4]=n+2*j;
            node[m+5]=n+4*j;
            node[m+6]=n+2*j-1;

            node[m+7]=n;
            node[m+8]=n+2;
            node[m+9]=n+4*j+1;
            node[m+10]=n+1;
            node[m+11]=n+2*j+1;
            node[m+12]=n+2*j;
        }
    }

    for(j=1;j<=ny;j++){
            n=(2*j-1)*j;
            m=j*j;
```

```
            node[6*(m-1)+1]=n;
            node[6*(m-1)+2]=n+4*j+1;
            node[6*(m-1)+3]=n+4*j-1;
            node[6*(m-1)+4]=n+2*j;
            node[6*(m-1)+5]=n+4*j;
            node[6*(m-1)+6]=n+2*j-1;
    }


    for (j=1;j<=ny1;j++){
        for (i=1;i<=j;i++){
            n=(j-1)*j/2+i;
            xv[n]=double(i-1)*x0/double(2*ny);
            yv[n]=double(j-1)*y0/double(2*ny);
        }
    }

    for (i=1;i<=nn;i++){
        iconv[i]=0;
        const1[i]=0;
        const2[i]=0.0;
    }

    n=ny*ny1;
    for (j=1;j<=ny1;j++){
        n1=j*(j-1)/2+1;
        n2=j+n;
        n3=j*(j+1)/2;

        const1[n1]=1;
        const1[n2]=1;
        const1[n3]=1;

        const2[n1]=0.0;
        const2[n2]=0.0;
        const2[n3]=0.0;
    }

    for(i=1;i<=ne;i++){
        Q[i]=10.0;
    }

    kx=1.0;ky=1.0;
}


void mesh_ed_3nte(double x0,double y0)//generates the mesh for an elliptic
                                      //domain by three node triangular elements
{
    long   i,j,m,n,i1,j1,k1,m1,n1,nr1,ntheta;
    double pi=3.141592653589793;
    double sc,a,b,r,theta;

    nr1=nr-1;
    m=-3;
    n=0;
    i1=1;
    for(i=1;i<=nr1;i++){
        ntheta=nc*i;
        for(j=1;j<=ntheta;j++){
            m+=3;
            n++;

            if(j==1) node[m+1]=i1;
            else if(div(j-1,i).rem==0) node[m+1]=node[m-2];
            else node[m+1]=node[m-2]+1;

            node[m+2]=n+1;
            node[m+3]=n+2;
            if(j==1){
                i1=node[m+1];
                k1=node[m+2];
```

```
            }
        }
        node[m+1]=i1;
        node[m+3]=k1;
        i1=k1;
    }


    m1=m;
    n1=n;
    j1=nc+3;
    for(i=2;i<=nr1;i++){
        ntheta=nc*(i-1);
        for(j=1;j<=ntheta;j++){
            m+=3;
            n++;

            if(j==1) node[m+2]=j1;
            else if(div(j-1,i-1).rem==0) node[m+2]=node[m-1]+2;
            else node[m+2]=node[m-1]+1;

            node[m+1]=n-n1+1;
            node[m+3]=node[m+1]+1;
            if(j==1) k1=node[m+1];
        }
        node[m+3]=k1;
        j1=node[m+2]+2;
    }


    xv[1]=0;
    yv[1]=0;
    i1=1;
    for(i=1;i<=nr1;i++){
        sc=double(i)/double(nr1);
        ntheta=nc*i;
        for(j=1;j<=ntheta;j++){
            i1++;
            theta=2.0*pi*double(j-1)/double(ntheta);
            a=sin(theta);
            b=cos(theta);
            r=sc*x0*y0/sqrt(x0*x0*a*a+y0*y0*b*b);
            xv[i1]=r*b;
            yv[i1]=r*a;
        }
    }


    for (i=1;i<=nn;i++){
        iconv[i]=0;
        const1[i]=0;
        const2[i]=0.0;
    }


    i1=nc*(nr-1);
    n=nc*(nr-1)*(nr-2)/2+1;
    for(i=1;i<=i1;i++){
        n++;
        const1[n]=1;
        const2[n]=0.0;
    }

    for(i=1;i<=ne;i++){
        Q[i]=10.0;
    }

    kx=1.0;ky=1.0;
}


void bandwidth()//calculates the height of each individual column for
                //skyline format
```

```
{
    long i,j,num,n1,a;
    long n[npe+1];

    n1=nn+1;
    for(i=1;i<=n1;i++){
        maxh[i]=0;
    }

    for(num=1;num<=ne;num++){

        i=npe*(num-1);
        for(j=1;j<=npe;j++){
            n[j]=node[i+j];
        }

        for(i=1;i<=npe;i++){
            for(j=1;j<=npe;j++){
                if(n[i]>n[j]){
                    n1=n[i]+1;
                    a=n[i]-n[j]+1;
                }
                else{
                    n1=n[j]+1;
                    a=n[j]-n[i]+1;
                }
                if(maxh[n1]<a) maxh[n1]=a;
            }
        }
    }

    maxh[1]=1;
    maxh[2]=2;
    n1=nn+1;
    for(i=3;i<=n1;i++){
        maxh[i]+=maxh[i-1];
    }

}

void core_3nte(char type)//calculates element matrices and assembles them in
                         //the global matrix for three node triangular elements
{
    long       i,j,ii,num;
    double     area,s,Tinfm,hm,qm,conv;
    long       n[npe+1];
    double     b[npe+1],c[npe+1],x[npe+1],y[npe+1];
    double     k1[npe+1][npe+1],k2[npe+1][npe+1],p1[npe+1],p2[npe+1],p3[npe+1];

    for (num=1;num<=ne;num++){

        for (i=1;i<=npe;i++){
            p1[i]=0.0;
            p2[i]=0.0;
            p3[i]=0.0;
            for(j=1;j<=npe;j++){
                k1[i][j]=0.0;
                k2[i][j]=0.0;
            }
        }

        i=npe*(num-1);
        for(j=1;j<=npe;j++){
            n[j]=node[i+j];
            x[j]=xv[n[j]];
            y[j]=yv[n[j]];
        }

        b[1]=y[3]-y[2];     b[2]=y[1]-y[3];     b[3]=y[2]-y[1];
        c[1]=x[2]-x[3];     c[2]=x[3]-x[1];     c[3]=x[1]-x[2];

        area=fabs( ((x[1]-x[3])*(y[2]-y[3])-(x[2]-x[3])*(y[1]-y[3])) /2.0);
```

```
for(i=1;i<=npe;i++){
    for(j=i;j<=npe;j++){
        k1[i][j]=(kx*b[i]*b[j]+ky*c[i]*c[j])/(4.0*area);
    }
}

for (i=1;i<=npe;i++){
    p1[i]=Q[num]*area/3.0;
}


if ((iconv[n[1]]+iconv[n[2]])==2){
        s=sqrt( (x[1]-x[2])*(x[1]-x[2])+(y[1]-y[2])*(y[1]-y[2]) );
        Tinfm=(Tinf[n[1]]+Tinf[n[2]])/2;
        hm=(h[n[1]]+h[n[2]])/2;
        qm=(q[n[1]]+q[n[2]])/2;
        conv=hm*s/6.0;

        k2[1][1]=2.0*conv;
        k2[1][2]=conv;
        k2[2][2]=k2[1][1];

        p2[1]=hm*Tinfm*s/2.0;
        p2[2]=p2[1];

        p3[1]=qm*s/2.0;
        p3[2]=p3[1];
}

if ((iconv[n[2]]+iconv[n[3]])==2){
        s=sqrt( (x[2]-x[3])*(x[2]-x[3])+(y[2]-y[3])*(y[2]-y[3]) );
        Tinfm=(Tinf[n[2]]+Tinf[n[3]])/2;
        hm=(h[n[2]]+h[n[3]])/2;
        qm=(q[n[2]]+q[n[3]])/2;
        conv=hm*s/6.0;

        k2[2][2]=2.0*conv;
        k2[2][3]=conv;
        k2[3][3]=k2[2][2];

        p2[2]=hm*Tinfm*s/2.0;
        p2[3]=p2[2];

        p3[2]=qm*s/2.0;
        p3[3]=p3[2];
}

if ((iconv[n[3]]+iconv[n[1]])==2){
        s=sqrt( (x[3]-x[1])*(x[3]-x[1])+(y[3]-y[1])*(y[3]-y[1]) );
        Tinfm=(Tinf[n[3]]+Tinf[n[1]])/2;
        hm=(h[n[3]]+h[n[1]])/2;
        qm=(q[n[3]]+q[n[1]])/2;
        conv=hm*s/6.0;

        k2[1][1]=2.0*conv;
        k2[1][3]=conv;
        k2[3][3]=k2[1][1];

        p2[1]=hm*Tinfm*s/2.0;
        p2[3]=p2[1];

        p3[1]=qm*s/2.0;
        p3[3]=p3[1];
}


switch(type){
    case 'b' :
        for(i=1;i<=npe;i++){
            p[n[i]]+=p1[i]+p2[i]+p3[i];
            for(j=i;j<=npe;j++){
                if(n[i]>n[j]) ii=(n[j]-1)*nb+n[i]-n[j]+1;
                else          ii=(n[i]-1)*nb+n[j]-n[i]+1;
```

```
                        gk[ii]+=k1[i][j]+k2[i][j];
                    }
                }
                break;
            case 's' :
                for(i=1;i<=npe;i++){
                    p[n[i]]+=p1[i]+p2[i]+p3[i];
                    for(j=i;j<=npe;j++){
                        if(n[i]>n[j]) ii=maxh[n[i]]+n[i]-n[j];
                        else          ii=maxh[n[j]]+n[j]-n[i];
                        gk[ii]+=k1[i][j]+k2[i][j];
                    }
                }
                break;
        }
    }

}

void core_6nte(char type)//calculates element matrices and assembles them in
                         //the global matrix for six node triangular elements
{
    long       i,j,ii,num;
    double     J11,J12,J21,J22,area,a11,a12,a22,s,Tinfm,hm,qm,conv;
    long       n[npe+1];
    double     x[npe+1],y[npe+1],k1[npe+1][npe+1],k2[npe+1][npe+1];
    double     p1[npe+1],p2[npe+1],p3[npe+1];

    for (num=1;num<=ne;num++){

        for (i=1;i<=npe;i++){
            p1[i]=0.0;
            p2[i]=0.0;
            p3[i]=0.0;
            for(j=1;j<=npe;j++){
                k1[i][j]=0.0;
                k2[i][j]=0.0;
            }
        }

        i=npe*(num-1);
        for(j=1;j<=npe;j++){
            n[j]=node[i+j];
            x[j]=xv[n[j]];
            y[j]=yv[n[j]];
        }

        J11=x[1]-x[3];     J12=y[1]-y[3];
        J21=x[2]-x[3];     J22=y[2]-y[3];

        area=fabs( (J11*J22-J12*J21) /2.0);

        a11=(kx*J22*J22+ky*J21*J21)/2.0/area;
        a12=-(kx*J22*J12+ky*J21*J11)/2.0/area;
        a22=(kx*J12*J12+ky*J11*J11)/2.0/area;

        k1[1][1]=0.5*a11;
        k1[1][2]=-a12/6.0;
        k1[1][3]=(a11+a12)/6.0;
        k1[1][4]=2.0/3.0*a12;
        k1[1][5]=0.0;
        k1[1][6]=-2.0/3.0*(a11+a12);
        k1[2][2]=0.5*a22;
        k1[2][3]=(a12+a22)/6.0;
        k1[2][4]=2.0/3.0*a12;
        k1[2][5]=-2.0/3.0*(a12+a22);
        k1[2][6]=0.0;
        k1[3][3]=0.5*(a11+2*a12+a22);
        k1[3][4]=0.0;
        k1[3][5]=-2.0/3.0*(a12+a22);
        k1[3][6]=-2.0/3.0*(a11+a12);
        k1[4][4]=4.0/3.0*(a11+a12+a22);
        k1[4][5]=-4.0/3.0*(a11+a12);
```

```
k1[4][6]=-4.0/3.0*(a12+a22);
k1[5][5]=4.0/3.0*(a11+a12+a22);
k1[5][6]=4.0/3.0*a12;
k1[6][6]=4.0/3.0*(a11+a12+a22);

for (i=4;i<=npe;i++){
    p1[i]=Q[num]*area/3.0;
}

if ((iconv[n[1]]+iconv[n[2]])==2){
        s=sqrt( (x[1]-x[2])*(x[1]-x[2])+(y[1]-y[2])*(y[1]-y[2]) );
        Tinfm=(Tinf[n[1]]+Tinf[n[4]]+Tinf[n[2]])/2;
        hm=(h[n[1]]+h[n[4]]+h[n[2]])/2;
        qm=(q[n[1]]+q[n[4]]+q[n[2]])/2;
        conv=hm*s/30.0;

        k2[1][1]=4.0*conv;
        k2[1][2]=-conv;
        k2[1][4]=2.0*conv;
        k2[2][2]=k2[1][1];
        k2[2][4]=k2[1][4];
        k2[4][4]=16.0*conv;

        p2[1]=hm*Tinfm*s/6.0;
        p2[2]=p2[1];
        p2[4]=2.0*p2[1];

        p3[4]=qm*s/3.0;
}

if ((iconv[n[2]]+iconv[n[3]])==2){
        s=sqrt( (x[2]-x[3])*(x[2]-x[3])+(y[2]-y[3])*(y[2]-y[3]) );
        Tinfm=(Tinf[n[2]]+Tinf[n[5]]+Tinf[n[3]])/2;
        hm=(h[n[2]]+h[n[5]]+h[n[3]])/2;
        qm=(q[n[2]]+q[n[5]]+q[n[3]])/2;
        conv=hm*s/30.0;

        k2[2][2]=4.0*conv;
        k2[2][3]=-conv;
        k2[2][5]=2.0*conv;
        k2[3][3]=k2[2][2];
        k2[3][5]=k2[2][5];
        k2[5][5]=16.0*conv;

        p2[2]=hm*Tinfm*s/6.0;
        p2[3]=p2[2];
        p2[5]=2.0*p2[2];

        p3[5]=qm*s/3.0;
}

if ((iconv[n[3]]+iconv[n[1]])==2){
        s=sqrt( (x[3]-x[1])*(x[3]-x[1])+(y[3]-y[1])*(y[3]-y[1]) );
        Tinfm=(Tinf[n[3]]+Tinf[n[6]]+Tinf[n[1]])/2;
        hm=(h[n[3]]+h[n[6]]+h[n[1]])/2;
        qm=(q[n[3]]+q[n[6]]+q[n[1]])/2;
        conv=hm*s/30.0;

        k2[1][1]=4.0*conv;
        k2[1][3]=-conv;
        k2[1][6]=2.0*conv;
        k2[3][3]=k2[1][1];
        k2[3][6]=k2[1][6];
        k2[6][6]=16.0*conv;

        p2[1]=hm*Tinfm*s/6.0;
        p2[3]=p2[1];
        p2[6]=2.0*p2[1];

        p3[6]=qm*s/3.0;
}
```

```
        switch(type){
            case 'b' :
                for(i=1;i<=npe;i++){
                    p[n[i]]+=p1[i]+p2[i]+p3[i];
                    for(j=i;j<=npe;j++){
                        if(n[i]>n[j]) ii=(n[j]-1)*nb+n[i]-n[j]+1;
                        else          ii=(n[i]-1)*nb+n[j]-n[i]+1;
                        gk[ii]+=k1[i][j]+k2[i][j];
                    }
                }
                break;
            case 's' :
                for(i=1;i<=npe;i++){
                    p[n[i]]+=p1[i]+p2[i]+p3[i];
                    for(j=i;j<=npe;j++){
                        if(n[i]>n[j]) ii=maxh[n[i]]+n[i]-n[j];
                        else          ii=maxh[n[j]]+n[j]-n[i];
                        gk[ii]+=k1[i][j]+k2[i][j];
                    }
                }
                break;
        }
    }

}

void boundary_sym_banded(long neqns,long nbw,double mat[],double rhs[],
                         long cond1[],double cond2[])
//imposes the boundary conditions on primary variables for
//a banded symmetric form
{
    long ii,i1,i2,i3,j;

    for (ii=1;ii<=neqns;ii++){

        if (cond1[ii]!=0){
            i1=ii;
            i2=ii;
            i3=(ii-1)*nbw;
            for (j=2;j<=nbw;j++){
               i1--;
               i2++;
               if(i1>=1){
                   rhs[i1]-=mat[(i1-1)*nbw+j]*cond2[ii];
                   mat[(i1-1)*nbw+j]=0.0;
               }
               if(i2<=neqns){
                   rhs[i2]-=mat[i3+j]*cond2[ii];
                   mat[i3+j]=0.0;
               }
            }

            rhs[ii]=cond2[ii];
            mat[i3+1]=1.0;
        }
    }

}

void boundary_sym_skyline(long neqns,long nbw,double mat[],double rhs[],
                          long maxh[],long cond1[],double cond2[])
//imposes the boundary conditions on primary variables for
//a skyline symmetric form
{
    long ii,i1,i2,j,j1,nbw1;

    nbw1=nbw-1;
    for (ii=1;ii<=neqns;ii++){

        if (cond1[ii]!=0){
            i1=ii;
            i2=maxh[ii];
            j1=maxh[ii+1]-maxh[ii]-1;
```

```
            for(j=1;j<=j1;j++){
                i1--;
                rhs[i1]-=mat[i2+j]*cond2[ii];
                mat[i2+j]=0.0;
            }
            i1=ii;
            for(j=1;j<=nbw1;j++){
                i1++;
                if(i1<=neqns && j<=maxh[i1+1]-maxh[i1]-1){
                    rhs[i1]-=mat[maxh[i1]+j]*cond2[ii];
                    mat[maxh[i1]+j]=0.0;
                }
            }

            rhs[ii]=cond2[ii];
            mat[i2]=1.0;
        }
    }

}
void solve_gauss_sym_banded(long neqns,long nbw,double mat[],double rhs[])
//solves the system of equations in banded symmetric form by
//Gauss elimination method
{

    long        i,j,ii,jj,m,n,n1,n2,n3;
    double      factor;

    m=neqns-1;
    for(n=1;n<=m;n++){
        n1=n+1;
        n2=n+nbw-1;
        n3=(n-1)*nbw;
        if (n2>neqns) n2=neqns;
        for(i=n1;i<=n2;i++){
            j=i-n+1;
            factor=mat[n3+j]/mat[n3+1];
            for(j=i;j<=n2;j++){
                ii=j-i+1;
                jj=j-n+1;
                mat[(i-1)*nbw+ii]-=factor*mat[n3+jj];
            }
            rhs[i]-=factor*rhs[n];
        }
    }


    // Back substitution

    for(n=neqns;n>=2;n--){
        n1=n-1;
        n2=n-nbw+1;
        rhs[n]/=mat[(n-1)*nb+1];
        if (n2<1) n2=1;
        for(i=n1;i>=n2;i--){
            j=n-i+1;
            rhs[i]-=mat[(i-1)*nbw+j]*rhs[n];
        }
    }

    rhs[1]/=mat[1];
}

void solve_LUdecom_sym_skyline(long neqns,double mat[],double rhs[],long maxh[])
//solves the system of equations in skyline symmetric form by
//LU decomposition method
{
    long   n,m,m1,m2,m3,m4,i,i1,i2,i3,ii,j,k;
    double a,b;

    for(n=1;n<=neqns;n++){
        m=maxh[n];
        m1=m+1;
```

```
        m2=maxh[n+1]-1;
        m3=m2-m1;
        if(m3>0){
            i=n-m3;
            i1=0;
            m4=m2;
            for(j=1;j<=m3;j++){
                i1++;
                m4--;
                i2=maxh[i];
                i3=maxh[i+1]-i2-1;
                if(i3>0){
                    ii=min(i1,i3);
                    a=0.0;
                    for(k=1;k<=ii;k++){
                        a+=mat[i2+k]*mat[m4+k];
                    }
                    mat[m4]-=a;
                }
                i++;
            }
        }
        if(m3>=0){
            i=n;
            b=0.0;
            for(ii=m1;ii<=m2;ii++){
                i--;
                i2=maxh[i];
                a=mat[ii]/mat[i2];
                b+=a*mat[ii];
                mat[ii]=a;
            }
            mat[m]-=b;
        }
    }

    //Reduation of right-hand-side veator

    for(n=1;n<=neqns;n++){
        m1=maxh[n]+1;
        m2=maxh[n+1]-1;
        if(m2-m1>=0){
            i=n;
            a=0.0;
            for(ii=m1;ii<=m2;ii++){
                i--;
                a+=mat[ii]*rhs[i];
            }
            rhs[n]-=a;
        }
    }

    //back substitution

    for(n=1;n<=neqns;n++){
        i=maxh[n];
        rhs[n]/=mat[i];
    }
    n=neqns;
    for(k=2;k<=neqns;k++){
        m1=maxh[n]+1;
        m2=maxh[n+1]-1;
        if(m2-m1>=0){
            i=n;
            for(ii=m1;ii<=m2;ii++){
                i--;
                rhs[i]-=mat[ii]*rhs[n];
            }
        }
        n--;
    }
}
```

```
void file_out()
//writes the results on some files
{
    long i,n;
    n=npe*ne;

    ofstream out1;
    out1.open("out1.dat");
    for (i=1;i<=nn;i++){
        out1<<p[i]<<"  ";
    }

    ofstream out2;
    out2.open("out2.dat");
    for (i=1;i<=n;i++){
        out2<<node[i]<<"  ";
    }

    ofstream out3;
    out3.open("out3.dat");
    for (i=1;i<=nn;i++){
        out3<<xv[i]<<"  ";
    }

    ofstream out4;
    out4.open("out4.dat");
    for (i=1;i<=nn;i++){
        out4<<yv[i]<<"  ";
    }

}
```

```
/*************************************************************************
*
*                       TWO DIMENSIONAL STATIC ELASTICITY
*
*
*                                   C++ Code
*                               Version: January 2009
*        Author: Arash Zamani, PhD from AmirKabir University of Technology
*
*
*             This code has been tested by Microsoft Visual C++ 2008
*************************************************************************
*
*                       Introducing global variables
*
* nx=number of divisions in x direction
* ny=number of divisions in y direction
* npe=number of nodes per element
* nb=bandwith
* ne=number of elements
* nn=number of nodes
* ndf=number of degrees of freedom per node
* ndfe=number of degrees of freedom per element
* tndf=total number of degrees of freedom
* node=connectivity
* xv=vector of x values
* yv=vector of y values
* const1=determines whether to impose primary boundary conditions
* const2=determines the value of primary boundary conditions
* gk=global stiffness matrix
* p=the vector which is first used to store external forces and then
*   to store the calculated primary variables from finite element model
* landa,G=Lame constants
* ro=density
*
*************************************************************************/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <io.h>
#include <iostream>
#include <fstream>
#include <float.h>
using namespace std;

//four node rectangular element (LRE)
const long  nx=24,ny=6,npe=4,ndf=2;
const long  nb=ndf*(nx+3),nn=(nx+1)*(ny+1);

//eight node rectangular element (serendipity)
//const long  nx=12,ny=2,npe=8,ndf=2;
//const long  nb=ndf*(3*nx+5),nn=(3*nx+2)*(ny+1)-nx-1;

const long  ne=nx*ny,tndf=ndf*nn,ndfe=npe*ndf;
long        node[npe*ne+1],const1[tndf+1];
double      xv[nn+1], yv[nn+1],const2[tndf+1];
double      gk[tndf*nb+1],p[tndf+1];
double      G=77.0*pow(10,9),landa=114.0*pow(10,9),ro=7820.0;


void mesh_LRE(double x0,double y0);
void mesh_serendipity(double x0,double y0);
void core_LRE();
void core_serendipity();
void boundary_sym_banded(long neqns,long nbw,double mat[],double rhs[],
                         long cond1[],double cond2[]);
void solve_gauss_sym_banded(long neqns,long nbw,double mat[],double rhs[]);
void file_out();
```

```
void main(void)
{

   mesh_LRE(6.0,1.0);
   core_LRE();
   boundary_sym_banded(tndf,nb,gk,p,const1,const2);
   solve_gauss_sym_banded(tndf,nb,gk,p);

   file_out();
}

void mesh_LRE(double x0,double y0)//generates the mesh for a rectangular
                                  //domain by four node rectangular elements
{
    long i,j,n,m,nx1,ny1;
    double f;

    nx1=nx+1;
    ny1=ny+1;
    m=-4;
    for (j=1;j<=ny;j++){
        for (i=1;i<=nx;i++){

            m+=4;
            n=nx1*(j-1)+i;

            node[m+1]=n;
            node[m+2]=n+1;
            node[m+3]=n+nx+2;
            node[m+4]=n+nx+1;
        }
    }

    for (i=1;i<=nx1;i++){
        for (j=1;j<=ny1;j++){
            n=nx1*(j-1)+i;
            xv[n]=double(i-1)*x0/double(nx);
            yv[n]=double(j-1)*y0/double(ny);
        }
    }

    for (i=1;i<=tndf;i++){
        const1[i]=0;
    }

    for (j=1;j<=ny1;j++){
        n=nx1*(j-1)+1;

        const1[ndf*(n-1)+1]=1;
        const2[ndf*(n-1)+1]=0.0;

        const1[ndf*(n-1)+2]=1;
        const2[ndf*(n-1)+2]=0.0;
    }

    f=-1.0/double(ny);
    for (j=2;j<=ny;j++){
        n=nx1*j;
        p[ndf*(n-1)+2]=f;
    }
    p[ndf*(nx1-1)+2]=f/2.0;
    p[ndf*(nx1*ny1-1)+2]=f/2.0;
}

void mesh_serendipity(double x0,double y0)//generates the mesh for a rectangular
                                          //domain by eight node rectangular
                                          //elements
{
    long   i,j,n,m,nx1,ny1;
    double f,dx,dy;
```

```
nx1=nx+1;
ny1=ny+1;

dx=x0/double(2*nx);
dy=y0/double(2*ny);

m=-8;
for (j=1;j<=ny;j++){
    for (i=1;i<=nx;i++){

        m+=8;
        n=(3*nx+2)*(j-1)+2*i-1;

        node[m+1]=n;
        node[m+2]=n+1;
        node[m+3]=n+2;
        node[m+4]=n+2*nx+3-i;
        node[m+5]=n+3*nx+4;
        node[m+6]=node[m+5]-1;
        node[m+7]=node[m+5]-2;
        node[m+8]=node[m+4]-1;
    }
}

for (i=1;i<=nx1;i++){
    for (j=1;j<=ny1;j++){
        n=(3*nx+2)*(j-1)+2*i-1;
        xv[n]=double(2*i-2)*dx;
        yv[n]=double(2*j-2)*dy;
    }
}

for (i=1;i<=nx;i++){
    for (j=1;j<=ny1;j++){
        n=(3*nx+2)*(j-1)+2*i;
        xv[n]=double(2*i-1)*dx;
        yv[n]=double(2*j-2)*dy;
    }
}

for (i=1;i<=nx1;i++){
    for (j=1;j<=ny;j++){
        n=(3*nx+2)*(j-1)+2*nx+1+i;
        xv[n]=double(2*i-2)*dx;
        yv[n]=double(2*j-1)*dy;
    }
}

for (i=1;i<=tndf;i++){
    const1[i]=0;
}

for (j=1;j<=ny1;j++){
    n=(3*nx+2)*(j-1)+1;

    const1[ndf*(n-1)+1]=1;
    const2[ndf*(n-1)+1]=0.0;

    const1[ndf*(n-1)+2]=1;
    const2[ndf*(n-1)+2]=0.0;
}

for (j=1;j<=ny;j++){
    n=(3*nx+2)*(j-1)+(2*nx+1)+1;

    const1[ndf*(n-1)+1]=1;
    const2[ndf*(n-1)+1]=0.0;

    const1[ndf*(n-1)+2]=1;
    const2[ndf*(n-1)+2]=0.0;
}
```

```
    f=-1.0/double(2*ny);
    for(j=2;j<=ny;j++){

        n=(3*nx+2)*(j-1)+2*nx+1;
        p[ndf*(n-1)+2]=f;

        n=(3*nx+2)*j;
        p[ndf*(n-1)+2]=f;
    }
    n=2*nx+1;
    p[ndf*(n-1)+2]=f/2.0;

    n=3*nx+2;
    p[ndf*(n-1)+2]=f;

    n=(3*nx+2)*ny+2*nx+1;
    p[ndf*(n-1)+2]=f/2.0;

}

void core_LRE()//calculates element matrices and assembles them in
               //the global matrix for four node rectangular elements
{
   long   i,j,ii,jj,ig,jg,r1,r2,num;
   double c00,c01,c02,c11,c12,c22,a,b;
   long   n[npe+1];
   double x[npe+1],y[npe+1];
   double k11[npe+1][npe+1],k12[npe+1][npe+1],k22[npe+1][npe+1];
   double k1[ndfe+1][ndfe+1],p1[ndfe+1];

   double S11[4][4]={{2,-2,-1,1},{-2,2,1,-1},{-1,1,2,-2},{1,-1,-2,2}};
   double S12[4][4]={{1,1,-1,-1},{-1,-1,1,1},{-1,-1,1,1},{1,1,-1,-1}};
   double S22[4][4]={{2,1,-1,-2},{1,2,-2,-1},{-1,-2,2,1},{-2,-1,1,2}};


   for (num=1;num<=ne;num++){

       ii=npe*(num-1);
       for (j=1;j<=npe;j++){

           n[j]=node[ii+j];
           x[j]=xv[n[j]];
           y[j]=yv[n[j]];
       }

       a=x[2]-x[1];
       b=y[4]-y[1];


       c00=a*b/36.0;
       c01=b/12.0;
       c02=a/12.0;
       c11=b/6.0/a;
       c12=1.0/4.0;
       c22=a/6.0/b;

       for (i=1;i<=npe;i++){
           for (j=1;j<=npe;j++){

               k11[i][j]=(2.0*G+landa)*c11*S11[i-1][j-1]+G*c22*S22[i-1][j-1];
               k12[i][j]=G*c12*S12[j-1][i-1]+landa*c12*S12[i-1][j-1];
               k22[i][j]=(2.0*G+landa)*c22*S22[i-1][j-1]+G*c11*S11[i-1][j-1];
           }
       }


       ii=1;
       for (i=1;i<=npe;i++){
           jj=1;
           for (j=1;j<=npe;j++){

               k1[ii][jj]=k11[i][j];
               k1[ii][jj+1]=k12[i][j];
```

```
                k1[ii+1][jj]=k12[j][i];
                k1[ii+1][jj+1]=k22[i][j];
                jj+=ndf;
            }
            ii+=ndf;
        }


        for (i=1;i<=ndfe;i++) p1[i]=0.0;

        for (i=1;i<=npe;i++){
            ig=(n[i]-1)*ndf;
            r1=(i-1)*ndf;
            for (ii=1;ii<=ndf;ii++){
                ig++;r1++;
                p[ig]+=p1[r1];
                for (j=1;j<=npe;j++){
                     jg=(n[j]-1)*ndf;
                     r2=(j-1)*ndf;
                     for (jj=1;jj<=ndf;jj++){
                         jg++;r2++;
                         if(jg>=ig)     gk[(ig-1)*nb+jg-ig+1]+=k1[r1][r2];
                     }
                }
            }
        }
    }

}

void core_serendipity()//calculates element matrices and assembles them in
                       //the global matrix for eight node rectangular elements
{
    long   i,j,ii,jj,ig,jg,r1,r2,num;
    double c00,c01,c02,c11,c12,c22,a,b;
    long   n[npe+1];
    double x[npe+1],y[npe+1];
    double k11[npe+1][npe+1],k12[npe+1][npe+1],k22[npe+1][npe+1];
    double k1[ndfe+1][ndfe+1],p1[ndfe+1];

    double S11[8][8]={{52, -80, 28, -6, 23, -40,17, 6},
                      {-80, 160, -80, 0, -40, 80, -40, 0},
                      {28, -80,52, 6, 17, -40, 23, -6},
                      {-6, 0, 6, 48, 6, 0, -6, -48},
                      {23, -40, 17, 6, 52, -80, 28, -6},
                      {-40, 80, -40, 0, -80, 160, -80, 0},
                      {17, -40, 23, -6, 28, -80, 52, 6},
                      {6, 0, -6, -48, -6, 0, 6, 48}};

    double S12[8][8]={{17, 4, -3, -4, 7, -4, 3, -20},
                      {-20, 0, 20, -16, -4, 0, 4, 16},
                      {3, -4, -17, 20, -3, 4, -7, 4},
                      {-4, -16, -4, 0, 4, 16, 4, 0},
                      {7, -4, 3, -20, 17, 4, -3, -4},
                      {-4, 0, 4, 16, -20, 0, 20, -16},
                      {-3, 4, -7, 4, 3, -4, -17, 20},
                      {4, 16, 4, 0, -4, -16, -4, 0}};

    double S22[8][8]={{52, 6, 17, -40, 23, -6, 28, -80},
                      {6, 48, 6, 0, -6, -48, -6, 0},
                      {17, 6, 52, -80, 28, -6, 23, -40},
                      {-40, 0, -80, 160, -80, 0, -40, 80},
                      {23, -6, 28, -80, 52, 6, 17, -40},
                      {-6, -48, -6, 0, 6, 48, 6, 0},
                      {28, -6, 23, -40, 17, 6, 52, -80},
                      {-80, 0, -40, 80, -40, 0, -80, 160}};



    for (num=1;num<=ne;num++){

        ii=npe*(num-1);
        for (j=1;j<=npe;j++){
```

```
            n[j]=node[ii+j];
            x[j]=xv[n[j]];
            y[j]=yv[n[j]];
        }

        a=x[3]-x[1];
        b=y[7]-y[1];

        c00=a*b/45.0;
        c01=b/90.0;
        c02=a/90.0;
        c11=b/a/90.0;
        c22=a/b/90.0;
        c12=1.0/36.0;

        for (i=1;i<=npe;i++){
            for (j=1;j<=npe;j++){

                k11[i][j]=(2.0*G+landa)*c11*S11[i-1][j-1]+G*c22*S22[i-1][j-1];
                k22[i][j]=(2.0*G+landa)*c22*S22[i-1][j-1]+G*c11*S11[i-1][j-1];
                k12[i][j]=G*c12*S12[j-1][i-1]+landa*c12*S12[i-1][j-1];
            }
        }


        ii=1;
        for (i=1;i<=npe;i++){
            jj=1;
            for (j=1;j<=npe;j++){

                k1[ii][jj]=k11[i][j];
                k1[ii][jj+1]=k12[i][j];
                k1[ii+1][jj]=k12[j][i];
                k1[ii+1][jj+1]=k22[i][j];
                jj+=ndf;
            }
            ii+=ndf;
        }


        for (i=1;i<=ndfe;i++)  p1[i]=0;

        for (i=1;i<=npe;i++){
            ig=(n[i]-1)*ndf;
            r1=(i-1)*ndf;
            for (ii=1;ii<=ndf;ii++){
                ig++;r1++;
                p[ig]+=p1[r1];
                for (j=1;j<=npe;j++){
                     jg=(n[j]-1)*ndf;
                     r2=(j-1)*ndf;
                     for (jj=1;jj<=ndf;jj++){
                         jg++;r2++;
                         if(jg>=ig)     gk[(ig-1)*nb+jg-ig+1]+=k1[r1][r2];
                     }
                }
            }
        }
    }

}

void boundary_sym_banded(long neqns,long nbw,double mat[],double rhs[],
                         long cond1[],double cond2[])
//imposes the boundary conditions on primary variables for
//a banded symmetric form
{
    long ii,i1,i2,i3,j;

    for (ii=1;ii<=neqns;ii++){

        if (cond1[ii]!=0){
```

```
            i1=ii;
            i2=ii;
            i3=(ii-1)*nbw;
            for (j=2;j<=nbw;j++){
               i1--;
               i2++;
               if(i1>=1){
                   rhs[i1]-=mat[(i1-1)*nbw+j]*cond2[ii];
                   mat[(i1-1)*nbw+j]=0.0;
               }
               if(i2<=neqns){
                   rhs[i2]-=mat[i3+j]*cond2[ii];
                   mat[i3+j]=0.0;
               }
            }

            rhs[ii]=cond2[ii];
            mat[i3+1]=1.0;
        }
    }

}

void solve_gauss_sym_banded(long neqns,long nbw,double mat[],double rhs[])
//solves the system of equations in banded symmetric form by
//Gauss elimination method
{

    long        i,j,ii,jj,m,n,n1,n2,n3;
    double      factor;

    m=neqns-1;
    for(n=1;n<=m;n++){
        n1=n+1;
        n2=n+nbw-1;
        n3=(n-1)*nbw;
        if (n2>neqns) n2=neqns;
        for(i=n1;i<=n2;i++){
            j=i-n+1;
            factor=mat[n3+j]/mat[n3+1];
            for(j=i;j<=n2;j++){
                ii=j-i+1;
                jj=j-n+1;
                mat[(i-1)*nbw+ii]-=factor*mat[n3+jj];
            }
            rhs[i]-=factor*rhs[n];
        }
    }


    // Back substitution

    for(n=neqns;n>=2;n--){
        n1=n-1;
        n2=n-nbw+1;
        rhs[n]/=mat[(n-1)*nb+1];
        if (n2<1) n2=1;
        for(i=n1;i>=n2;i--){
            j=n-i+1;
            rhs[i]-=mat[(i-1)*nbw+j]*rhs[n];
        }
    }

    rhs[1]/=mat[1];
}

void file_out()
//writes the results on a file
{
    long i,n;
    n=npe*ne;

    ofstream out1;
```

```
    out1.open("out1.dat");
    for (i=1;i<=tndf;i++){
        out1<<p[i]<<"  ";
    }

    ofstream out2;
    out2.open("out2.dat");
    for (i=1;i<=n;i++){
        out2<<node[i]<<"  ";
    }

    ofstream out3;
    out3.open("out3.dat");
    for (i=1;i<=nn;i++){
        out3<<xv[i]<<"  ";
    }

    ofstream out4;
    out4.open("out4.dat");
    for (i=1;i<=nn;i++){
        out4<<yv[i]<<"  ";
    }

}
```

```
/*****************************************************************************
*
*                          3D TRANSIENT HEAT CONDUCTION
*                  BY USING SPARSE MATRICES AND ITERATIVE SOLVER
*
*
*                                    C++ Code
*                                Version: August 2009
*
*        Author: Arash Zamani, PhD from AmirKabir University of Technology
*
*
*              This code has been tested by Microsoft Visual C++ 2008
*****************************************************************************/

#include <math.h>
#include <iostream>
#include <fstream>
#include <time.h>
#include <atltime.h>
#include <map>
#include <assert.h>
using namespace std;


//In the following, two classes are defined to use dynamic memory (heap segment)
//to store vectors of int and double types respectively. The lenght of the vectors
//can be defined in terms of nonconstant integers which are computed in running
//time of the code. The checking for out of range vector indices is made optional
//in these classes because it increases the computational time slightly.
//The default is to check the bounds:

#define vec_bounds_check

class veci
{
    protected:
        int    *p_;
        int    dim_;

    public:
        veci::veci(int n) :  p_(new int[n]), dim_(n)
        {
            assert( p_!=NULL );
            for (int i=0; i<n; i++)      p_[i] = 0;
        }

        veci::~veci()
        {
            if( p_ )   delete [] p_;
        }

        int&  operator[](int i){
            #ifdef vec_bounds_check
            assert(i < dim_);
            #endif
            return p_[i];
        }
};


class vecd
{
    protected:
        double *p_;
        int    dim_;

    public:
        vecd::vecd(int n) :  p_(new double[n]), dim_(n)
        {
            assert( p_!=NULL );
            for (int i=0; i<n; i++)      p_[i] = 0.0;
        }
```

```
        vecd::~vecd()
        {
            if( p_ )   delete [] p_;
        }

        double&  operator[](int i){
            #ifdef vec_bounds_check
            assert(i < dim_);
            #endif
            return p_[i];
        }
};

//////////////////////////////////////////////////////////////////////////////


//In the following, the data types used for finding nonzero entries of the sparse
//matrices are defined. The map class provided by the C++ standard template library
//is used for this purpose. The structure of a sparse matrix is found by defining
//a mapping from row and column indices of matrix nonzero entries onto the entries
//numbers "m". The number "m" is then used to locate entry values in a vector
//representing the matrix. The type "mappi" which maps a pair of int numbers on
//a single int number is defined to do this. Another type defined below, "mappd",
//is used to store the values of those matrix entries that will be changed due
//wiping rows and columns in the stage of imposing essential boundary conditions.

typedef   map< pair<int, int>, int >      mappi;
typedef   map< pair<int, int>, double >   mappd;


//////////////////////////////////////////////////////////////////////////////


double Det(double a11, double a12, double a13,
           double a21, double a22, double a23,
           double a31, double a32, double a33);

void  Mesh_Cube(double xL, double yL, double zL, int xdim, int ydim, int zdim,
                vecd& xv, vecd& yv, vecd& zv, veci& node, veci& nodeS);

void Core_LTE(int neqns, int ne, int neS, veci& node, veci& nodeS,  vecd& xv, vecd& yv,
              vecd& zv, vecd& matV, vecd& matA, mappi& Sp, vecd& kval, vecd& cval, vecd↙
    & h,
              double kx, double ky, double kz);

void Flux_LTE(int neqns, int ne, int neS, veci& node, veci& nodeS, vecd& xv, vecd& yv,
              vecd& zv, vecd& matV, vecd& matA, vecd& p, vecd& Q, vecd& Tinf, vecd& h, ↙
    vecd& q);

void Adjust(int jN, int neqns, int nz, veci& cond1, vecd& cond2,
            veci& rind, veci& cind, vecd& kval, vecd& rhs, mappd& Spcond);

void Preconditioner(int nz, veci& rind, veci& cind, vecd& kval,vecd& M);

int  Conjugate_Gradient(int max_iter, int& num_iter, double tol, int neqns, int nz,
                        veci& rind, veci& cind, vecd& kval, vecd& rhs, vecd& x, vecd&  ↙
    M);


//////////////////////////////////////////////////////////////////////////////


void main()
{


    const int  npe = 4;
    int        result, i, j, m, num, neqns, num_iter, ne, nz, neS, tnum, jN;
    double     bet, dt, T1, invdt, k1, c1, pt, coe, kx, ky, kz, orient;
    double     dxc1, dyc1, dzc1, dx12, dx13, dy12, dy13, dz12, dz13, a1, a2, a3;

    double     tol = pow(10.0,-8.0);//The tolerance for iterative solver.
```

```
veci       n(npe+1);
vecd       xn(npe+1), yn(npe+1), zn(npe+1);



clock_t    Time[8];
clock_t    TimeS[9];

Time[1] = clock();

//Defining material properties. They are nondimensionalized by the parameters
//"L" and "ve" which are the charactristic lenght and time respectively. It is
//further explained in "A Zamani, MR Eslami, Coupled dynamical thermoelasticity
//of a functionaly graded cracked layer, J. Thermal Stresses 2009; 32:969-985."

double  L=1.0, ve=0.00005;

double  TB   = 273.0;
double  kxP  = 17.0;
double  kyP  = 17.0;
double  kzP  = 17.0;
double  rhoP = 7833.0;
double  cP   = 461.0;

kx = kxP/(L*rhoP*cP*ve);
ky = kyP/(L*rhoP*cP*ve);
kz = kzP/(L*rhoP*cP*ve);

////////////////////////////////////////////////////////////////////////////

//Deciding whether to generate mesh for a cube by the subprogram "Mesh_Cube.cpp"
//or to read the mesh information from a file generated by gmesh software for
//a 3D arbitrary shape of the solution domain. The default is to mesh a cube
// by using the subprogram "Mesh_Cube.cpp".

//#define use_gmesh

////////////////////////////////////////////////////////////////////////////

//Reading the input data from "file_name.msh" if the gmesh software is used.

#ifdef use_gmesh

int n1, n2, n3, n4, it, itS, netotal, neqns1;
char *file_name="Dodecahedron.msh";
ifstream in1;
in1.open( file_name );
for(i=1; i<=4; i++)   in1.ignore(200,'\n');
in1>>neqns;

veci    cond1(neqns+1);
vecd    xv(neqns+1),    yv(neqns+1),   zv(neqns+1);
vecd    x(neqns+1),     p(neqns+1),    p0(neqns+1);
vecd    cond2(neqns+1), Tinf(neqns+1), h(neqns+1);

vecd    M(neqns+1);//The Jacobi preconditioner.


for(i=1; i<=neqns; i++){
    in1>>n1;
    in1>>xv[n1];
    in1>>yv[n1];
    in1>>zv[n1];
}

for(i=1; i<=3; i++)   in1.ignore(200,'\n');
in1>>netotal;

ne=0;
neS=0;
for(i=1; i<=netotal; i++){
    in1>>n1;
    in1>>n2;
```

```
    if( n2==4 )       ne++;
    else if( n2==2 )  neS++;
    in1.ignore(200,'\n');
}

veci  node(npe*ne+1), nodeS(3*neS+1);
vecd  matV(ne+1), matA(neS+1), Q(ne+1), q(neS+1);

in1.seekg( 0, ios_base::beg );
neqns1=neqns+8;
for(i=1; i<=neqns1; i++)      in1.ignore(200,'\n');

it=0;
itS=0;
for(i=1; i<=netotal; i++){
    in1>>n1;
    in1>>n2;
    if( n2==4 ){
        it++;
        in1>>n3;
        for(j=1; j<=n3; j++)  in1>>n4;

        for(j=1; j<=npe; j++)  in1>>node[npe*(it-1)+j];
    }
    else if( n2==2 ){
        itS++;
        in1>>n3;
        for(j=1; j<=n3; j++)  in1>>n4;

        for(j=1; j<=3; j++)  in1>>nodeS[3*(itS-1)+j];
    }

    else in1.ignore(200,'\n');
}
in1.close();

#endif

////////////////////////////////////////////////////////////////////////////

//Generating the mesh for a cube if the gmesh software is not used.

#ifndef use_gmesh

double  xL=2.0,   yL=1.0,   zL=1.5;
int     xdim=40,  ydim=20,  zdim=30;

neqns = (xdim+1)*(ydim+1)*(zdim+1);
ne    = 5*xdim*ydim*zdim;
neS   = 4*(xdim*ydim + ydim*zdim + zdim*xdim);

veci    cond1(neqns+1);
vecd    xv(neqns+1),    yv(neqns+1),   zv(neqns+1);
vecd    x(neqns+1),     p(neqns+1),    p0(neqns+1);
vecd    cond2(neqns+1), Tinf(neqns+1), h(neqns+1);

vecd    M(neqns+1);//The Jacobi preconditioner.

veci    node(npe*ne+1), nodeS(3*neS+1);
vecd    matV(ne+1), matA(neS+1), Q(ne+1), q(neS+1);

Mesh_Cube(xL, yL, zL, xdim, ydim, zdim, xv, yv, zv, node, nodeS);

#endif

////////////////////////////////////////////////////////////////////////////

//Checking the orientation of surface 2D triangles and correcting them if nessecary.

double xcen=0.0, ycen=0.0, zcen=0.0;
//The point A(xcen, ycen, zcen) is inside the body.

for(num=1; num<=neS; num++){
```

```
    i=3*(num-1);
    for(j=1; j<=3; j++){
        n[j]=nodeS[i+j];
        xn[j]=xv[n[j]];
        yn[j]=yv[n[j]];
        zn[j]=zv[n[j]];
    }

    dxc1=xn[1]-xcen;    dx12=xn[2]-xn[1];    dx13=xn[3]-xn[1];
    dyc1=yn[1]-ycen;    dy12=yn[2]-yn[1];    dy13=yn[3]-yn[1];
    dzc1=zn[1]-zcen;    dz12=zn[2]-zn[1];    dz13=zn[3]-zn[1];

    orient=Det(dxc1, dyc1, dzc1,
               dx12, dy12, dz12,
               dx13, dy13, dz13);

    if( orient<0 ){
        nodeS[i+1]=n[1];
        nodeS[i+2]=n[3];
        nodeS[i+3]=n[2];
    }
}

////////////////////////////////////////////////////////////////////////////

//Defining heat flux on the surface of the body.

double  xr=-1.0, yr=-1.0, zr=-1.0;
//The vector ray(xr, yr, zr) shows direction of the heat flux.

coe=sqrt(xr*xr+yr*yr+zr*zr);
double  q1 = 200.0/(rhoP*cP*ve*TB);
for(num=1; num<=neS; num++){
    i=3*(num-1);
    for(j=1; j<=3; j++){
        n[j]=nodeS[i+j];
        xn[j]=xv[n[j]];
        yn[j]=yv[n[j]];
        zn[j]=zv[n[j]];
    }

    dx12=xn[2]-xn[1];     dx13=xn[3]-xn[1];
    dy12=yn[2]-yn[1];     dy13=yn[3]-yn[1];
    dz12=zn[2]-zn[1];     dz13=zn[3]-zn[1];

    a1=dy12*dz13-dy13*dz12;
    a2=-(dx12*dz13-dx13*dz12);
    a3=dx12*dy13-dx13*dy12;

    orient=xr*a1+yr*a2+zr*a3;

    if( orient<0 )  q[num]=-q1*orient*coe/sqrt(a1*a1+a2*a2+a3*a3);
}


Time[2]  = clock();
TimeS[1] = Time[2]-Time[1];


////////////////////////////////////////////////////////////////////////////

//Finding sparse structure of the system matrices.

pair<int, int>  Pr;
mappi           Sp;
mappd           Spcond;
typedef  mappi::value_type    Sp_add;
mappi::iterator               itSp, itSpEnd;
pair<mappi::iterator, bool>   ins;


m=0;
for(num=1; num<=ne; num++){
```

```
    i=npe*(num-1);
    for(j=1; j<=npe; j++)  n[j]=node[i+j];

    for(i=1; i<=npe; i++){
        for(j=1; j<=npe; j++){

            Pr.first=n[i];
            Pr.second=n[j];

            ins=Sp.insert( Sp_add( Pr, m ) );

            if( ins.second )  m++;

        }
    }
}

nz=Sp.size();//number of nonzero entries in the system matrices.

veci   rind(nz), cind(nz);//These vectors store row and column indices of
                          //nonzero enties respectively.

vecd   kval(nz), cval(nz);//These vectors store values of nonzero entires of
                          //stiffness and damping matrices respectively.

itSp=Sp.begin();
itSpEnd=Sp.end();
for(; itSp!=itSpEnd; ++itSp){
    Pr=itSp->first;
    i=itSp->second;
    rind[i]=Pr.first;
    cind[i]=Pr.second;
}

Time[3]  = clock();
TimeS[2] = Time[3]-Time[2];

////////////////////////////////////////////////////////////////////////////

//Calculating stiffness and damping matrices.

Core_LTE(neqns,ne,neS,node,nodeS,xv,yv,zv,matV,matA,Sp,kval,cval,h,kx,ky,kz);
Sp.clear();

//Calculating the matrices for time integration.

tnum=80;//Number of time steps.
bet=0.5;
T1=1.0;
dt=T1/double(tnum);

vecd  tvec(tnum+1);
tvec[0]=0.0;
for(i=1; i<=tnum; i++){
   tvec[i]=tvec[i-1]+dt;
}

invdt = 1.0/dt;

for(m=0; m<nz; m++){
    k1 = kval[m];
    c1 = cval[m];

    kval[m] = invdt*c1+bet*k1;
    cval[m] = invdt*c1-(1-bet)*k1;
}

Time[4]  = clock();
TimeS[3] = Time[4]-Time[3];

////////////////////////////////////////////////////////////////////////////
```

```
//Calculating the right-hand side vector (force vector) at time=0.0

Flux_LTE(neqns,ne,neS,node,nodeS,xv,yv,zv,matV,matA,p0,Q,Tinf,h,q);


//Looping over time steps and calculating the response for each of them.

TimeS[4]=0;
TimeS[5]=0;
TimeS[6]=0;
TimeS[7]=0;
FILE *out_file;
cout<<" Looping over time steps... "<<endl;


for(jN=1; jN<=tnum; jN++){

    //Calculating the right-hand side vector (force vector).

    Flux_LTE(neqns,ne,neS,node,nodeS,xv,yv,zv,matV,matA,p,Q,Tinf,h,q);

    for(i=1; i<=neqns; i++){
            pt    = p[i];
            p[i]  = (1-bet)*p0[i]+bet*p[i];
            p0[i] = pt;
    }
    for(m=0; m<nz; m++){
        i=rind[m];
        j=cind[m];
        p[i] += cval[m]*x[j];
    }


    Time[5]   =  clock();
    TimeS[4] += Time[5]-Time[4];

    ////////////////////////////////////////////////////////////////////////

    //Imposing essential boundary conditions.

     Adjust(jN,neqns,nz,cond1,cond2,rind,cind,kval,p,Spcond);

    Time[6]   = clock();
    TimeS[5] += Time[6]-Time[5];

    ////////////////////////////////////////////////////////////////////////

    //Calculating the Jacobi preconditioner at the first time step.

    if( jN==1 )   Preconditioner(nz,rind,cind,kval,M);

    ////////////////////////////////////////////////////////////////////////

    //Solving system of linear equations by Conjuate gradient method.

    result=Conjugate_Gradient(500,num_iter,tol,neqns,nz,rind,cind,kval,p,x,M);
    assert( !result );

    Time[7]   = clock();
    TimeS[6] += Time[7]-Time[6];

    ////////////////////////////////////////////////////////////////////////

    //Writing the number of iterations on a file.

    fopen_s( &out_file, "out_iterations.dat" , (jN==1) ? "w" : "a" );
    fprintf_s( out_file, " Time step = %d\n", jN );
    fprintf_s( out_file, " Number of iterations = %d\n\n", num_iter );
    fclose( out_file );

    ////////////////////////////////////////////////////////////////////////

    //Writing the results on "file_name.msh". This file was firstly used in
```

```
    //the begining of the code to input mesh data from gmesh software. Now
    //the results for each time step is appended to this file. After that
    //the file is again read by gmsh to plot temperature variations in three
    //dimensions.

    #ifdef use_gmesh
    fopen_s( &out_file, file_name , "a" );
    fprintf_s( out_file, "$NodeData\n"  );
    fprintf_s( out_file, "1\n"  );
    fprintf_s( out_file, "\"Temperature\"\n"  );
    fprintf_s( out_file, "1\n"  );
    fprintf_s( out_file, "%g\n", tvec[jN]  );
    fprintf_s( out_file, "3\n"  );
    fprintf_s( out_file, "%d\n", jN  );
    fprintf_s( out_file, "1\n"  );
    fprintf_s( out_file, "%d\n", neqns  );
    for(i=1; i<=neqns; i++)
        fprintf_s( out_file, "%d %g\n", i, x[i] );
    fprintf_s( out_file, "$EndNodeData\n" );
    fclose( out_file );
    #endif

    ////////////////////////////////////////////////////////////////////////

    //Writing the results on the file "out_cube.dat" if the gmesh software
    //was not chosen as the mesh generator.

    #ifndef use_gmesh
    fopen_s( &out_file, "out_cube.dat" , (jN==1) ? "w" : "a" );
    for(i=1; i<=neqns; i++)
        fprintf_s( out_file, "%g\n", x[i] );
    fclose( out_file );
    #endif


    //Showing the progress through time steps.
    cout<<" Time step["<<jN<<"] was done."<<endl;


    Time[4]   = clock();
    TimeS[7] += Time[4]-Time[7];

}



//Writing size specifications of the model on a file.

fopen_s( &out_file, "out_size.dat" , "w");
fprintf_s( out_file, " Number of time steps = %d\n\n", tnum );
fprintf_s( out_file, " Number of degrees of freedom = %d\n\n", neqns );
fprintf_s( out_file, " Number of 3D tetrahedronal elements = %d\n\n", ne );
fprintf_s( out_file, " Number of 2D triangles on the surface = %d\n\n", neS );
fprintf_s( out_file, " Number of nonzero entries in the system matrices = %d", nz);
fclose( out_file );

////////////////////////////////////////////////////////////////////////////

//Writing the elapsed times for different segments of the computations on a file.
//The time format is "minutes : seconds". The resolution is one millisecond.

int mins[9];
double secs[9], time;

TimeS[8]=0;
for(i=1; i<=7; i++)   TimeS[8] += TimeS[i];

for(i=1; i<=8; i++){
    time    = (double) TimeS[i]/CLOCKS_PER_SEC;
    mins[i] = int(time/60.0);
    secs[i] = time-double(60*mins[i]);
}
```

```
    fopen_s( &out_file, "out_time.dat" , "w");
    fprintf_s( out_file, " Reading input data = mins: %d, secs: %g\n\n",
        mins[1], secs[1] );
    fprintf_s( out_file, " Finding sparse structure = mins: %d, secs: %g\n\n",
        mins[2], secs[2] );
    fprintf_s( out_file, " Computing the system matrices = mins: %d, secs: %g\n\n",
        mins[3], secs[3] );
    fprintf_s( out_file, " Computing the right-hand side vector = mins: %d, secs: %g\n\↙
    n",
        mins[4], secs[4] );
    fprintf_s( out_file, " Imposing essential boundary conditions = mins: %d, secs: %g\↙
    n\n",
        mins[5], secs[5] );
    fprintf_s( out_file, " Solving system of linear equations = mins: %d, secs: %g\n\n"↙
    ,
        mins[6], secs[6] );
    fprintf_s( out_file, " Writing the results = mins: %d, secs: %g\n\n",
        mins[7], secs[7] );
    fprintf_s( out_file, "================================================\n" );
    fprintf_s( out_file, " Total elapsed time = mins: %d, secs: %g",
        mins[8], secs[8] );
    fclose( out_file );


}


////////////////////////////////////////////////////////////////////////////


double Det(double a11, double a12, double a13,
           double a21, double a22, double a23,
           double a31, double a32, double a33)
//Calculates Deterninant of a 3*3 matrix.
{
    return  a11*(a22*a33-a32*a23)-a12*(a21*a33-a31*a23)+a13*(a21*a32-a31*a22);
}


////////////////////////////////////////////////////////////////////////////

void  Mesh_Cube(double xL, double yL, double zL, int xdim, int ydim, int zdim,
                vecd& xv, vecd& yv, vecd& zv, veci& node, veci& nodeS)
//Generates a tetrahedronal mesh for a cube.
 {

     int npe=4;

     int     i, j, k, ii, jj, TetNum, HexNum, start, p, n0, n7;
     double  x0, y0, z0, x1, y1, z1, x7, y7, z7;

     int   xdim1=xdim+1, ydim1=ydim+1, zdim1=zdim+1;
     int   neqns = xdim1*ydim1*zdim1;

     //Fist the cube is divided into hexahedronal (cubic) elements and then
     //each hexahedron will be divided into five tetrahedra.

     int   neH   = xdim*ydim*zdim;//Number of hexahedronal (cubic) elements.

     veci  nodeH(8*neH+1);//Connectivity for hexahedronal elements.

     vecd  con(neH+1);//Determines the orientation of tetrahedra in the hexahedron.



     double  dx = xL/double(xdim);
     double  dy = yL/double(ydim);
     double  dz = zL/double(zdim);

     double  tol = pow(10.0,-8.0);


     // Calculating the nodes coordinates
```

```
ii=0;
x0=-0.5*xL;
y0=-0.5*yL;
z0=-0.5*zL;

x1=x0;
y1=y0;
z1=z0;
for(k=1; k<=zdim1; k++){
    for(j=1; j<=ydim1; j++){
        for(i=1; i<=xdim1; i++){

            ii++;

            xv[ii] = x1;
            yv[ii] = y1;
            zv[ii] = z1;

            x1 += dx;
        }
        x1  = x0;
        y1 += dy;
    }
    y1  = y0;
    z1 += dz;
}


//Assigning global node numbers for hexahedrons

start=1;
for(i=1; i<=neH; i++){

    j=8*(i-1)+1;

    nodeH[j]   = start;
    nodeH[j+1] = nodeH[j]+1;
    nodeH[j+2] = nodeH[j]+xdim1;
    nodeH[j+3] = nodeH[j+2]+1;
    nodeH[j+4] = nodeH[j]+xdim1*ydim1;
    nodeH[j+5] = nodeH[j+4]+1;
    nodeH[j+6] = nodeH[j+4]+xdim1;
    nodeH[j+7] = nodeH[j+6]+1;

    if(  i%(ydim*xdim)==0  )  start += xdim+3;
    else                      start++;
    if(   start%xdim1==0   )  start++;
}



//Dividing each of the hexahedra into five tetrahedra

int vec1[20] = {0, 3, 2, 6,
                3, 5, 7, 6,
                0, 1, 3, 5,
                0, 6, 4, 5,
                0, 5, 3, 6};

int vec2[20] = {0, 1, 2, 4,
                1, 7, 4, 5,
                1, 7, 3, 2,
                2, 6, 4, 7,
                1, 2, 4, 7};


TetNum=-4;
for(k=0; k<zdim; k++){
    for(j=0; j<ydim; j++){
        for(i=0; i<xdim; i++){

            TetNum += 5;
```

```
            HexNum  = k*ydim*xdim + j*xdim + i + 1;

            con[HexNum] =
                pow(-1.0,(double)i)*pow(-1.0,(double)j)*pow(-1.0,(double)k);

            ii=npe*(TetNum-1)+1;
            jj=8*(HexNum-1)+1;

            if( con[HexNum]>0.0 )
                for(p=0; p<20; p++)   node[ii+p]=nodeH[jj+vec1[p]];
            else
                for(p=0; p<20; p++)   node[ii+p]=nodeH[jj+vec2[p]];


        }
    }
}


//Finding surface 2D triangles

int  vec1x0[6] = {0, 4, 6, 0, 6, 2};
int  vec1y0[6] = {0, 1, 5, 0, 5, 4};
int  vec1z0[6] = {0, 2, 3, 0, 3, 1};

int  vec1x7[6] = {7, 5, 3, 1, 3, 5};
int  vec1y7[6] = {7, 3, 6, 2, 6, 3};
int  vec1z7[6] = {7, 6, 5, 6, 4, 5};



int  vec2x0[6] = {0, 4, 2, 2, 4, 6};
int  vec2y0[6] = {0, 1, 4, 1, 5, 4};
int  vec2z0[6] = {0, 2, 1, 1, 2, 3};

int  vec2x7[6] = {7, 5, 1, 7, 1, 3};
int  vec2y7[6] = {7, 2, 6, 7, 3, 2};
int  vec2z7[6] = {7, 6, 4, 7, 4, 5};


x0=-0.5*xL;
y0=-0.5*yL;
z0=-0.5*zL;

x7=0.5*xL;
y7=0.5*yL;
z7=0.5*zL;

k=0;
for(i=1; i<=neH; i++){

    j=8*(i-1)+1;
    n0=nodeH[j];
    n7=nodeH[j+7];


    if( con[i]>0.0 ){

    if( fabs(xv[n0]-x0)<tol ) for(p=0; p<6; p++)  nodeS[++k]=nodeH[j+vec1x0[p]];
    if( fabs(yv[n0]-y0)<tol ) for(p=0; p<6; p++)  nodeS[++k]=nodeH[j+vec1y0[p]];
    if( fabs(zv[n0]-z0)<tol ) for(p=0; p<6; p++)  nodeS[++k]=nodeH[j+vec1z0[p]];
    if( fabs(xv[n7]-x7)<tol ) for(p=0; p<6; p++)  nodeS[++k]=nodeH[j+vec1x7[p]];
    if( fabs(yv[n7]-y7)<tol ) for(p=0; p<6; p++)  nodeS[++k]=nodeH[j+vec1y7[p]];
    if( fabs(zv[n7]-z7)<tol ) for(p=0; p<6; p++)  nodeS[++k]=nodeH[j+vec1z7[p]];

    }
    else{

    if( fabs(xv[n0]-x0)<tol ) for(p=0; p<6; p++)  nodeS[++k]=nodeH[j+vec2x0[p]];
    if( fabs(yv[n0]-y0)<tol ) for(p=0; p<6; p++)  nodeS[++k]=nodeH[j+vec2y0[p]];
    if( fabs(zv[n0]-z0)<tol ) for(p=0; p<6; p++)  nodeS[++k]=nodeH[j+vec2z0[p]];
    if( fabs(xv[n7]-x7)<tol ) for(p=0; p<6; p++)  nodeS[++k]=nodeH[j+vec2x7[p]];
    if( fabs(yv[n7]-y7)<tol ) for(p=0; p<6; p++)  nodeS[++k]=nodeH[j+vec2y7[p]];
```

```
         if( fabs(zv[n7]-z7)<tol ) for(p=0; p<6; p++)  nodeS[++k]=nodeH[j+vec2z7[p]];

         }
      }


      //Writing the nodes coordinates on a file

      FILE *out_file;
      fopen_s( &out_file, "out_xyz.dat" , "w" );
      for(i=1; i<=neqns; i++)
            fprintf_s( out_file, "%g %g %g\n", xv[i], yv[i], zv[i] );
      fclose( out_file );


}
////////////////////////////////////////////////////////////////////////////////

void Core_LTE(int neqns, int ne, int neS, veci& node, veci& nodeS,  vecd& xv, vecd& yv,
              vecd& zv, vecd& matV, vecd& matA, mappi& Sp, vecd& kval, vecd& cval, vecd↙
    & h,
              double kx, double ky, double kz)
//Calculates stiffness and damping matrices.
{

    const int  npe = 4;
    int    i, j, k, m, ii, num;
    double  coe, coe1, coe2, Ve, a1, a2, a3;
    double  Aijk, hm;
    double  k1[npe+1][npe+1], k2[npe+1][npe+1], c1[npe+1][npe+1];
    veci    n(npe+1);
    vecd    x(npe+1), y(npe+1), z(npe+1), b(npe+1), c(npe+1), d(npe+1), v(npe+1);


    int arrange[7]={1,2,3,4,1,2,3};

    pair<int, int> Pr;
    mappi::iterator   itSp;


    for (num=1;num<=ne;num++){


        i=npe*(num-1);
        for(j=1;j<=npe;j++){
            n[j]=node[i+j];
            x[j]=xv[n[j]];
            y[j]=yv[n[j]];
            z[j]=zv[n[j]];
        }


        for(ii=1; ii<=4; ii++){

            i=arrange[ii-1];
            j=arrange[ii];
            k=arrange[ii+1];
            m=arrange[ii+2];

            if( i==1 || i==3 ) coe=1.0;
            else               coe=-1.0;

            b[i]=coe*Det(1.0, y[j], z[j],
                         1.0, y[k], z[k],
                         1.0, y[m], z[m]);

            c[i]=coe*Det(x[j], 1.0, z[j],
                         x[k], 1.0, z[k],
                         x[m], 1.0, z[m]);

            d[i]=coe*Det(x[j], y[j], 1.0,
                         x[k], y[k], 1.0,
                         x[m], y[m], 1.0);
```

```
        v[i]=coe*Det(x[j],y[j],z[j],
                     x[k],y[k],z[k],
                     x[m],y[m],z[m]);


    }


    Ve=(v[1]+v[2]+v[3]+v[4])/6.0;


    matV[num]=Ve;


    coe1=1.0/(36.0*Ve);
    coe2=Ve/20.0;
    for(i=1; i<=npe; i++){
        for(j=1; j<=npe; j++){
            k1[i][j] = coe1*(kx*b[i]*b[j]+ky*c[i]*c[j]+kz*d[i]*d[j]);
            c1[i][j] = coe2;
        }
        c1[i][i] += coe2;
    }



    for(i=1; i<=npe; i++){
        for(j=1; j<=npe; j++){

            Pr.first=n[i];
            Pr.second=n[j];
            itSp=Sp.find( Pr );
            m=itSp->second;

            kval[m] += k1[i][j];
            cval[m] += c1[i][j];

        }
    }
}



////////////////////////////////////////////////////////////////////////



for (num=1;num<=neS;num++){

    i=3*(num-1);
    for(j=1;j<=3;j++){
        n[j]=nodeS[i+j];
        x[j]=xv[n[j]];
        y[j]=yv[n[j]];
        z[j]=zv[n[j]];
    }


    a1=Det(1.0, y[1], z[1],
           1.0, y[2], z[2],
           1.0, y[3], z[3]);

    a2=Det(x[1], 1.0, z[1],
           x[2], 1.0, z[2],
           x[3], 1.0, z[3]);

    a3=Det(x[1], y[1], 1.0,
           x[2], y[2], 1.0,
           x[3], y[3], 1.0);


    Aijk=0.5*sqrt( a1*a1+a2*a2+a3*a3 );
```

```
        matA[num]=Aijk;

        hm=(h[n[1]]+h[n[2]]+h[n[3]])/2;

        coe=hm*Aijk/12.0;
        k2[1][1]=2.0*coe;  k2[1][2]=coe;      k2[1][3]=coe;
        k2[2][1]=coe;      k2[2][2]=2.0*coe;  k2[2][3]=coe;
        k2[3][1]=coe;      k2[3][2]=coe;      k2[3][3]=2*coe;



        for(i=1; i<=3; i++){
            for(j=1; j<=3; j++){

                Pr.first=n[i];
                Pr.second=n[j];
                itSp=Sp.find( Pr );
                m=itSp->second;

                kval[m] += k2[i][j];
            }
        }
    }



}


//////////////////////////////////////////////////////////////////////////////


void Flux_LTE(int neqns, int ne, int neS, veci& node, veci& nodeS, vecd& xv, vecd& yv,
              vecd& zv, vecd& matV, vecd& matA, vecd& p, vecd& Q, vecd& Tinf, vecd& h, ↙
    vecd& q)
//Calculates the right-hand side vector (force vector).
{

    const int   npe = 4;
    int     i, j, num;
    double  coe, Tinfm, hm, qm;
    veci    n(npe+1);
    vecd    p1(npe+1), p2(npe+1), p3(npe+1);


    for(i=1; i<=neqns; i++)   p[i] = 0.0;


    for (num=1;num<=ne;num++){

        i=npe*(num-1);
        for(j=1;j<=npe;j++)    n[j]=node[i+j];

        coe=matV[num]/4.0;
        for(i=1; i<=npe; i++)  p1[i]=coe*Q[num];

        for(i=1; i<=npe; i++)  p[n[i]] += p1[i];
    }



    //////////////////////////////////////////////////////////////////////////////



    for (num=1;num<=neS;num++){

        i=3*(num-1);
        for(j=1;j<=3;j++)   n[j]=nodeS[i+j];

        Tinfm=(Tinf[n[1]]+Tinf[n[2]]+Tinf[n[3]])/3.0;
        hm=(h[n[1]]+h[n[2]]+h[n[3]])/3.0;
```

```
        qm=q[num];

        coe=hm*Tinfm*matA[num]/3.0;
        p2[1]=coe;
        p2[2]=coe;
        p2[3]=coe;

        coe=qm*matA[num]/3.0;
        p3[1]=coe;
        p3[2]=coe;
        p3[3]=coe;

        for(i=1; i<=3; i++)     p[n[i]] += p2[i]+p3[i];
    }


}


////////////////////////////////////////////////////////////////////////////


void Adjust(int jN, int neqns, int nz, veci& cond1, vecd& cond2,
            veci& rind, veci& cind, vecd& kval, vecd& rhs, mappd& Spcond)
//Imposes essential boundary conditions.
{

    int i, j, m;

    pair<int, int>    Pr;
    mappd::iterator   itc, itcEnd;
    typedef  mappd::value_type  Spcond_Add;


    if( jN==1 ){

        for(m=0; m<nz; m++){

            i=rind[m];
            j=cind[m];
            if( cond1[i]!=0 ){
                if( i==j )  kval[m] = 1.0;
                else        kval[m] = 0.0;
            }
            else if( cond1[j]!=0 ){
                Pr.first=i;
                Pr.second=j;
                Spcond.insert( Spcond_Add(Pr, kval[m]) );
                rhs[i] -= kval[m]*cond2[j];
                kval[m] = 0.0;
            }
        }
    }
    else{

        itc=Spcond.begin();
        itcEnd=Spcond.end();
        for(; itc!=itcEnd; ++itc){

            Pr=itc->first;
            i=Pr.first;
            j=Pr.second;

            rhs[i] -= (itc->second)*cond2[j];
        }
    }



    for(i=1; i<=neqns; i++)   if( cond1[i]!=0 )  rhs[i]=cond2[i];


}
```

```
////////////////////////////////////////////////////////////////////////////

void Preconditioner(int nz, veci& rind, veci& cind, vecd& kval,vecd& M)
//Calculates the Jacobi preconditioner.
{
    int i,j, m;

    //Calculating the Jacobi preconditioner, "M". This is the simplest preconditioner.
    //More sophisticated ones, like incomplete Cholesky preconditioner, are more
    //efficient. It is possible to use Jacobi preconditioner of a matrix without
    //any additional storage beyond that of the matrix itself. But in the following,
    //the reciprocal of Jacobi preconditioner is stored in the vector "M". In this way,
    //the computational cost of repeative divisions in the solver, can be avoided.

    for(m=0; m<nz; m++){
        i=rind[m];
        j=cind[m];
        if( i==j )  M[i] = 1.0/kval[m];
    }

}

////////////////////////////////////////////////////////////////////////////

int  Conjugate_Gradient(int max_iter, int& num_iter, double tol, int neqns, int nz,
                        veci& rind, veci& cind, vecd& kval, vecd& rhs, vecd& x, vecd& M)
//Solves the system of linear equations by using Conjuate gradient method.
{
  int   i, j, iter, m;
  double dot, normb, normr, alpha, beta, rho, rho_1, resid;

  vecd   p(neqns+1), z(neqns+1), q(neqns+1), r(neqns+1);

  normb=0.0;
  for(i=1; i<=neqns; i++)   normb+=rhs[i]*rhs[i];
  normb = sqrt(normb);
  if( normb==0.0 ) normb=1.0;

  for(i=1; i<=neqns; i++)    r[i]=rhs[i];
  for(m=0; m<nz; m++){
      i=rind[m];
      j=cind[m];
      r[i] -= kval[m]*x[j];
  }



  for(iter=1; iter<=max_iter; iter++){

      rho=0.0;
      for(i=1; i<=neqns; i++){
          z[i]=M[i]*r[i];
          rho+=r[i]*z[i];
      }

      if (iter == 1)
          for(i=1; i<=neqns; i++)  p[i]=z[i];
      else {

          beta=rho/rho_1;
          for(i=1; i<=neqns; i++)  p[i]=z[i]+beta*p[i];
      }

      for(i=1; i<=neqns; i++)  q[i]=0.0;
      for(m=0; m<nz; m++){
          i=rind[m];
          j=cind[m];
          q[i] += kval[m]*p[j];
      }

      dot=0.0;
```

```
      for(i=1; i<=neqns; i++)  dot+=p[i]*q[i];
      alpha=rho/dot;

      normr=0.0;
      for(i=1; i<=neqns; i++){
          x[i]+=alpha*p[i];
          r[i]-=alpha*q[i];
          normr+=r[i]*r[i];
      }
      normr=sqrt(normr);
      resid=normr/normb;

      if(  resid <= tol  ){
          tol=resid;
          num_iter=iter;
          return 0;
      }
      rho_1=rho;
  }


  tol=resid;
  num_iter=max_iter;
  return 1;
}
```