A Description of *basicCube*

C. Godsalve
F2 Swainstone Road
Reading
RG2 ODX
United Kingdom
Tel: (0118) 9864389

email:seagods@hotmail.com

1 *basicCube*

The program *basicCube* generates a tetrahedral mesh on a "regular" rectangular grid, its purpose is to provide an input mesh to some other program. In particular it is designed to provide an initial mesh which may be deformed and altered so the mesh will conform to some arbitrary surface defining a material boundary within the initial mesh. That is we output the initial mesh so as it can be changed to a new mesh. This new mesh may represent complicated boundaries between different material regions for instance. Not only the node positions of the initial mesh provided by *basicCube* will be changed, but edges may be collapsed or new new nodes may be introduced. For this reason, *basicCube* does not take advantage of it's regularity to save on either hard-drive storage for data or RAM. It is to have the same kind of storage as for a general mesh.

As an example, in regular mesh we may easily find all the tetrahedra attached to a given node, so there is no need for the node to hold any data to enable us to find the tetrahedra attached to the node (apart from the mesh array dimensions). However, in a subsequent alteration of the mesh, such encoding may be lost. For this reason we encode as much information as may be needed for an irregular mesh.

One application may be, for instance, the approximate solutions for partial differential equations, or partial integro-differential equations. An example of the latter is the radiative transfer equation. For radiative transfer applications, we will need to trace rays through the mesh to calculate the first collision source term. To facilitate this, among other things, the mesh tetrahedra hold data that inform them of which other tetrahedra they share facets with.

The output of basicCube may also be used as input for a program to find triangulations for the isosurfaces of a function of three variables. Such a program will be an easy generalisation of finding the contours lines of a function of two variables. If $z = f(x, y)$ is defined on a triangulation of the (x, y) plane, then the contours are calculated from the intersections of the triangle's three (x, y, z) edges with a $z = z_c$ plane for different values of z_c. The base

triangle in the (x, y) plane is mapped onto a two dimensional triangle embedded in a three dimensional space via the values of the function at the corners.

The generalisation of this process is to have a function $w = f(x, y, z)$ defined at the nodes of the tetrahedral decomposition of the (x, y, z) space and find the intersection of the six (w, x, y, z) edges of a three dimensional tetrahedron embedded in a four dimensional space, and the $w = w_c$ hyperplane.

We may wish to use the same isosurface visualisation program to look at isosurfaces on a more general mesh, with the data resulting from some finite element calculation on that mesh. If the isosurface visualisation program that uses the *basicCube* mesh is to be the same as that that visualises isosurfaces of data on a general finite element mesh, then the *basicCube* output should not be specialised to regular rectangular grid.

To sum up, the purpose of *basicCube* is to provide an initial mesh for some other program. At present the output mesh mesh is in fact a regular parallelepiped.

2 The Mesh Structure

We start off with a Regular Rectangular Parallelepiped Grid (RRPG) of $isx \times isy \times isz$ nodes. We shall denote the number of grid spacings or gaps between the nodes as $igx = isx - 1$, $igy = isy - 1$, $igz = isz - 1$. Along the x direction, the x coordinates of the nodes shall be at $x_0, x_1, x_2, x_3, x_4, \ldots, x_{igx}$, and a similar notation is used for y and z. There is a bounding parallelepiped (BP) with corners at $(0, 0, 0)$, $(x_{igx}, 0, 0)$, $(x_{igx}, y_{igy}, 0)$, $(0, y_{igy}, 0)$, and $(0, 0, z_{igz})$, $(x_{igx}, 0, z_{igz})$, $(x_{igx}, y_{isy}, z_{igz})$, and (x_0, y_{igy}, z_{igz}).

We shall decompose this grid into a tetrahedral mesh, and output the mesh data to a file called "basicCube.dat". To allow the possibility of periodic boundaries we *must* insist that the isx, isy, isz are odd numbers if the periodic boundaries are in the x, y, or z directions. The reason why they must be odd numbers in this case will soon become clear.

We consider Sub-Parallelepipeds (SP)s with corners at (ixn, iyn, izn), and whose other corners have nodes at (ixm, iym, izm) where m is either n or $n + 1$. These of course, are the smallest possible SPs. Each of these SPs shall have the same node numbering system. This system is depicted in Fig.1. Now, the SP with a corner at $(0, 0, 0)$ shall be defined as $EVEN$, any SP that shares a face with an $EVEN$ SP shall be defined as ODD, and any SP sharing a face with an ODD SP shall be $EVEN$. In either case we may split up the SP into 5 tetrahedra, four shall have the first node at a corner of a the SP, and there shall be a fifth internal tetrahedron with no faces on the surface of the SP.

Every facet of every tetrahedron within the mesh *must* precisely match a facet of another tetrahedron. We call the vector from node 1 to node 2 $\mathbf{a}$, the vector from node 1 to node 3 $\mathbf{b}$, and the vector from node 1 to node 4 $\mathbf{c}$. For the applications in mind, we must arrange the mesh such that $\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$ is always positive, and $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ form a right handed basis with the origin at node 1.

At this point, we can see that $EVEN/ODD$ system allows us to match faces in neighbouring SPs. If we have periodic boundary conditions, then this system means we must have an even number of gaps to maintain face matching in the periodic system, hence the odd numbers for the number of nodes in the directions that have periodicity.

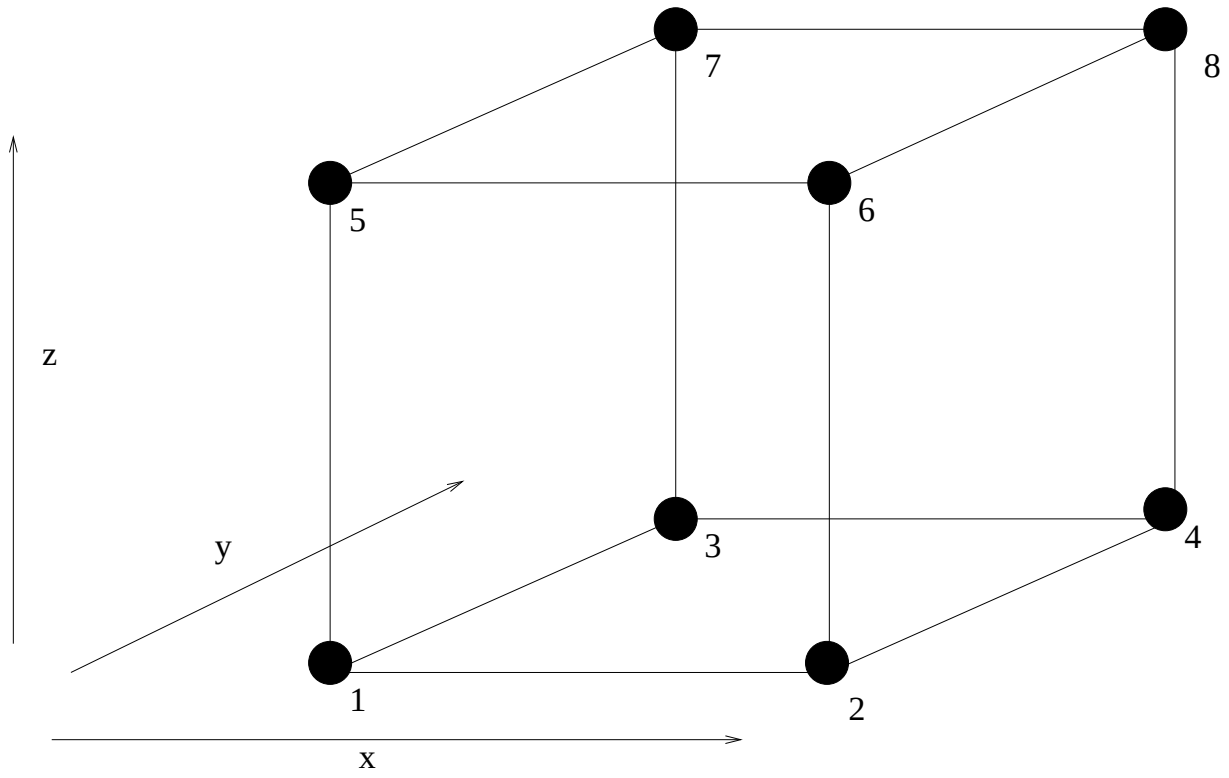

Figure 1: *The numbering of the nodes in an SP, with nodes 1, 2, 5, and 6 making up the facet facing the viewer.*

This gives rise to the node numbering scheme in the table below. The first four tetrahedra in for $EVEN$ and ODD SPs have internal coordinate systems as depicted in Fig.2.

TET No	TYPE	NODE 1	NODE 2	NODE 3	NODE 4
1	EVEN	1	2	3	5
2	EVEN	4	3	2	8
3	EVEN	6	5	8	2
4	EVEN	7	8	5	3
5	EVEN	2	3	5	8
2	ODD	2	4	1	6
1	ODD	3	1	4	7
3	ODD	5	7	6	1
4	ODD	8	6	7	4
5	ODD	1	4	7	6

Each tetrahedron has an origin at node 1, and **a** points to node 2, **b**, points to node 3, and **c** points to node 4. In this scheme, we call the facet with 3 nodes that do not include node 1, face 1, and so on. That is the facets are numbered according to the node not in the facet. In Fig.2, we can see how we can cross from one tetrahedron into another. We tabulate this below.

TET No	TYPE	FACE	TET No	FACE	TYPE	FACE	TET No	FACE
1	EVEN	1	5	4	ODD	1	5	3
1	EVEN	2	1	3	ODD	2	2	3
1	EVEN	3	2	2	ODD	3	1	2
1	EVEN	4	3	4	ODD	4	3	4
2	EVEN	1	5	3	ODD	1	5	4
2	EVEN	2	2	3	ODD	2	1	3
2	EVEN	3	1	2	ODD	3	2	2
2	EVEN	4	4	4	ODD	4	4	4
3	EVEN	1	5	2	ODD	1	5	2
3	EVEN	2	3	3	ODD	2	4	3
3	EVEN	3	4	2	ODD	3	3	2
3	EVEN	4	1	4	ODD	4	1	4
4	EVEN	1	5	1	ODD	1	5	1
4	EVEN	2	4	3	ODD	2	3	3
4	EVEN	3	3	2	ODD	3	4	2
4	EVEN	4	2	4	ODD	4	2	4
5	EVEN	1	4	1	ODD	1	4	4
5	EVEN	2	3	1	ODD	2	3	3
5	EVEN	3	2	1	ODD	3	1	1
5	EVEN	4	1	1	ODD	4	2	2

From the table, we read that if we are in an $EVEN$ SP, a point on face 3 of tetrahedron 1 is a point in tetrahedron 2 of the neighbouring SP and it is on face 2 of that tetrahedron. Likewise if we are in tetrahedron 4 of an ODD SP, and on face 3, then we are in tetrahedron 4 of the neighbouring SP and on face 2 of that tetrahedron. In all cases, face 1 of tetrahedra

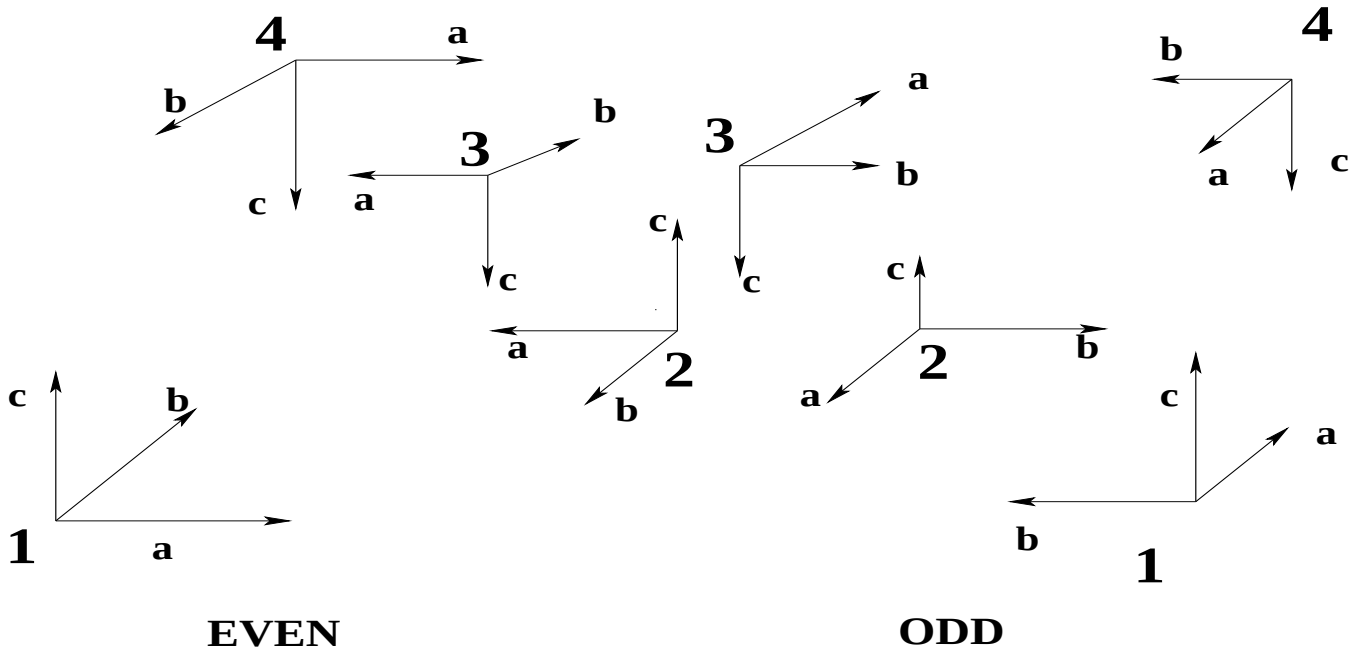

Figure 2: Coordinate systems for tetrahedra in $EVEN$ and ODD SPs.

1, 2, 3, and 4, crosses into tetrahedron 5 of the same SP, as this tetrahedron is the internal tetrahedron.

We have mentioned already that every tetrahedron has an origin (node 1) and three basis vectors **a**, **b**, and **c**. The coordinates of any point in this system are denoted α, β and γ. Face 2 corresponds to $\alpha = 0$, face 3 corresponds to $\beta = 0$ and face 4 corresponds to $\gamma = 0$. Face 1 corresponds to $\alpha + \beta + \gamma = 1$. A point on face 2 $(0, \beta, \gamma)$ has coordinates $(\alpha', 0, \gamma'$ on the face in the neighbouring SP where it is obvious that $\alpha' = \beta$ and $\gamma' = \gamma$. The situation is similar for faces 2 and 3. A shared point simply has α and β swapped as we change from one local coordinate system to another.

Shared faces for tetrahedron 5 are a little more complicated. A point on face 1 of any of tetrahedra 1, 2, 3, and 4 is at $\alpha\mathbf{a} + \beta\mathbf{b} + (1 - \alpha - \beta)\mathbf{c}$. The corresponding position vectors in tetrahedron 5 are different, and we tabulate them below. The inverse relations are easily found.

If we had for instance, a regular mesh, and with optical properties defined at the node positions, ray tracing is trivial. We have rectangular Cartesian coordinates for four out of five tetrahedra, and only two direction vectors to figure out in tetrahedron 5 for the $EVEN$ and ODD SP's. However, if the mesh has been deformed to accommodate cloud surfaces say, then in general, we have a different non rectangular coordinate system for each tetrahedron. If we have enough memory, we can store the direction calculate the direction vector for the first collision source term for each tetrahedron. Otherwise we have to work the vector out on arriving at each new tetrahedron. If we want to do the whole problem by ray-tracing, then we must store the three vectors per tetrahedron, with the (α, β, γ) coordinates for the Cartesian **i**

TET No	TYPE	$\mathbf{r}'$	α	β
1	EVEN	$\alpha'\,\mathbf{a}' + \beta'\,\mathbf{b}'$	$1 - \alpha' - \beta'$	α'
2	EVEN	$\alpha'\,\mathbf{a}' + \gamma'\,\mathbf{c}'$	α'	$1 - \alpha' - \gamma'$
3	EVEN	$\beta'\,\mathbf{b}' + \gamma'\,\mathbf{c}'$	β'	γ'
4	EVEN	$\alpha'\mathbf{a}' + \beta'\,\mathbf{b}' + (1 - \alpha' - \beta')\mathbf{c}'$	$1 - \alpha' - \beta'$	β'
1	ODD	$\alpha'\mathbf{a}' + \gamma'\mathbf{c}'$	α'	$\alpha' + \beta' - 1$
2	ODD	$\alpha'\mathbf{a}' + \beta'\mathbf{b}'$	$1 - \alpha' - \beta'$	α'
3	ODD	$\beta'\mathbf{b}' + \gamma'\mathbf{c}'$	β'	α'
4	ODD	$\alpha'\mathbf{a}' + \beta'\,\mathbf{b}' + (1 - \alpha' - \beta')\mathbf{c}'$	β'	$1 - \alpha' - \beta'$

Given all the above, we simply write a triple loop, fill in the node positions, and add in tetrahedra for each new "cube" in (i, j, k) space. The tetrahedron numbers in the cubes for $(i \pm 1, j, k)$, $(i, j \pm 1, k)$ and $(i, j, k \pm 1)$ can readily be found, and so we can store which tetrahedron face crosses into whichever face of which tetrahedron crossed into. We can then take into account whether we have periodic boundaries for those facets that coincide with the bounding parallelepiped (BP) for the entire mesh. We shall not go into any further details of the arithmetic here.

3 The RayMeshTet and MeshNode Objects

In an object oriented approach, one might expect to build tetrahedra from facet objects, and facet objects from edge objects and edge objects from node objects. However, in terms

RAM for a 3D mesh, this approach takes up more memory than is necessary.

So, we shall have a tetrahedron object of the class $RayMeshTet$. This shall store for node integers, and four pointers to integer which shall store the tetrahedron numbers found on crossing faces one to four. They are stored as pointers so that the neighbour on the facet can be $NULL$. Neighbours shall be $NULL$ if the facet is on the BP and there is no periodic condition on that face. On top of this we store four characters which serve as integers $c1$ to $c4$. These tell us that, if we cross face 1 into tetrahedron $*n1$, then we are on face $c1$ in that tetrahedron.

We also store a single bool for use in algorithms that find all the tetrahedra attached to a node, and to propagate an isosurface triangulation from how the particular isosurface crosses one tetrahedron. This is necessary in order to maintain the global orientation of the surface along with the orientation of a local triangle.

The reader might object that there is no need for an algorithm to find all the tetrahedra attached to a node. Indeed, in the mesh as described so far, there is no need. We can work them out quickly just from the node number. However, it should be recalled that this is an *initial* mesh. It may be later transformed by edge splitting or collapsing, edge swapping, and so on. That is to say that once the initial mesh has been converted into the mesh for solving the problem, we may no longer be able to know the tetrahedra surrounding each node just from the node number. We must however, demand that any process that alters the mesh shall maintain the fidelity of shared tetrahedron faces as new tetrahedra or and nodes are inserted, or old tetrahedra or nodes are deleted.

The $MeshNode$ object shall store a pointer to a $D3Dvec$ object which has various member functions and overloaded operators. For instance it has comparison functions useful for Oct-Trees. In addition to this it shall hold two integer characters $freedom$, $aliases$, and a pointer to integer ID that is initialised as $NULL$.

This ID pointer is not an ID for the node. The nodes are expected to be in an array, and the array counter shall serve as the the ID for the node. However, if we have periodic boundaries, the current node may also be matched to up to three other nodes. As the periodic boundaries are set, the number of these other nodes is found, which is stored in $aliases$. So, if one node is deleted or moved, so must is aliases be deleted in the same way. Note, that we have used this number to minimise the amount of memory stored. For ID only has the memory allocated for the number of aliases the node has, which is mostly zero. It is always zero if no periodic boundaries are set.

There is one important thing that should be remembered about these IDs. If, for instance, we have periodic boundaries in x, y, and z, then a function at the corners must be the same for all eight corners. However the corners only hold three IDs. To get all eight corners, we need to look at the aliases of the IDs held at one corner. This will get us only seven of corners. We need an alias of an alias of an alias to get at the last one. We may change this in future, but that's how it is at present.

The character variable $freedom$ holds information of how a can be moved. If $freedom = 0$ the node can be moved, and if $freedom = 1$ the node can be moved in any direction. After this $freedom$ takes on the values 2, 3, and 4 if it can only be moved in the x, y, or z directions, and 5, 6 and 7 if it can only be moved in the (x, y), (x, z) or (y, z) planes. If there is a bounding surface in the mesh, this too may have corners and edges. So, if $freedom$ has values 8, or 9 it is constrained to move on the surface, or an edge in the

surface. If it is on a corner in the surface, then freedom is set to zero, and it cannot move. Any intersection of the surface with the BP is to be regarded as a surface edge whether or not there are periodic boundaries.

4 Output Format

We shall give a brief description of the output format here. At the end, we shall provide code fragments for IO in C++.

The first line consists of two integers. These are the number of nodes and the number of tetrahedra. The next line consists of three bools, which are true or false as to whether we have periodic conditions in x, y, z. If these are all false, no alias data is output. Then, for each node, there will be a line of three floating point numbers and two integers. These are the x, y and z values of the node positions, a number of a tetrahedron that that particular node occurs in, and the value of the character representing the freedom of the node.

If any of the periodic conditions are set true, each line will have the x, y, and z coordinates, followed by a tetrahedron number, followed by the number of aliases for the node, followed by that number of aliases.

We note that there is no node identity number. It is assumed that the program reading them will identify the nodes as having numbers 0, to the number of nodes minus one. Any other encoding must be dealt with by the program reading the data. The same goes for the tetrahedra.

Following the nodes are the tetrahedra. Each tetrahedron has one line of 12 integers. The first four numbers are the node numbers of the tetrahedron. The next four are the numbers of the neighbouring tetrahedron on facets 1 to 4. If any of these are minus 1, then there is no neighbouring tetrahedron on that facet. The next numbers encode which face of the neighbouring tetrahedron is entered on leaving the facet of the current tetrahedron. That is, if we leave the current tetrahedron through facet two, we enter the neighbour's facet three, for instance. These numbers are not numbered the usual C way in that they are 1,2,3, or 4. They are initialised at zero in the tetrahedron's constructor function, so if one of these is zero, it has not been assigned. The tetrahedron's corresponding pointer to a neighbour was still NULL, and -1 was output for the neighbouring tetrahedron.

We finish with the following code fragments for input and output. The output section of *basicCube* is

```
    ofstream  file_out;
    file_out.open("basicCube.dat");
    file_out << inode << " " << itets << endl;
    file_out << periodicx <<" " << periodicy << " " << periodicz << endl;
    bool periodic;
    char charstop;
    if(periodicx || periodicy || periodicz)periodic=true;
    for(int i=0; i<inode; i++){
file_out <<NodeV[i].GetX() << " " << NodeV[i].GetY() << "  " <<NodeV[i].GetZ();
   file_out << " " << Node[i].GetMyTet() <<" " << (int)Node[i].GetFreedom();
```

```
        if(periodic){
        charstop=Node[i].GetAlias();
        file_out << "  " << (int)charstop;
        int* TempID=Node[i].GetID();
        for(int ichar=0; ichar<(int)charstop;ichar++){
           file_out << " " << (int)*(TempID+ichar);}
            file_out << endl;}
     }

     int n1,n2,n3, n4; //remember GetN returns a pointer so as it can be NULL
     int *p1,*p2,*p3,*p4;

     for(int i=0; i<itets;i++){
        file_out << (Tets+i)->Get1() << " " << (Tets+i)->Get2()
             << " " << (Tets+i)->Get3() << " " << (Tets+i)->Get4();

p1=(Tets+i)->GetN1();p2=(Tets+i)->GetN2();p3=(Tets+i)->GetN3();p4=(Tets+i)->GetN4();

        if(p1){n1=*p1;} else {n1=-1;}
        if(p2){n2=*p2;} else {n2=-1;}
        if(p3){n3=*p3;} else {n3=-1;}
        if(p4){n4=*p4;} else {n4=-1;}
        file_out << " " << n1 << " " << n2 <<" " << n3<< " " << n4;
 file_out << " " << (int)((Tets+i)->GetC1()) << " " << (int)((Tets+i)->GetC2())
    << " " << (int)((Tets+i)->GetC3()) << " " << (int)((Tets+i)->GetC4()) << endl;
     }
     file_out.close();
```

and the input fragment of datawrite (which reads in the mesh, and evaluates a function at each node, then outputs the mesh for the isosurface program) is

```
   file_in >> inodes >> itets;
   file_in >> periodicx >> periodicy >> periodicz;

   double x, y, z;
   int ifree,mytet;
   int n_aliases;
   int alias[3];

   D3Dvec* NodeVec;  RayMeshTet* Tets;  MeshNode* Nodes;
   NodeVec=new D3Dvec[inodes]; Tets=new RayMeshTet[itets]; Nodes=new MeshNode[inodes];

   if(periodicx || periodicy || periodicz)periodic=true;

   if(periodic){
      for(int i=0; i< inodes; i++){
      file_in >> x >> y >> z >> mytet >> ifree >> n_aliases;
      Nodes[i].SetPos(x,y,z,NodeVec+i);
```

```
      Nodes[i].SetMyTet(mytet);
      Nodes[i].SetFreedom(ifree);
      for(int ia=0; ia<n_aliases; ia++){
          file_in >> alias[ia];
          }
      if(n_aliases==1)Nodes[i].SetID(alias[0]);
      if(n_aliases==2)Nodes[i].SetID(alias[0], alias[1]);
      if(n_aliases==3)Nodes[i].SetID(alias[0], alias[1], alias[2]);
      }
   }
   else
   {
      for(int i=0; i< inodes; i++){
      file_in >> x >> y >> z >> mytet >> ifree;
      Nodes[i].SetPos(x,y,z,NodeVec+i);
      //meshnode holds NodeVec address only, it sets NodeVec to x,y,z
      Nodes[i].SetMyTet(mytet);
      Nodes[i].SetFreedom(ifree);
      }
   }
   //now read in tetrahedra
    int i1,i2,i3,i4,j1,j2,j3,j4,c1,c2,c3,c4;
    for(int i=0; i<itets;i++){
file_in >> i1 >> i2 >> i3 >> i4 >> j1 >> j2 >> j3 >> j4 >> c1 >> c2 >> c3 >> c4;
Tets[i].SetTet(i1,i2,i3,i4,j1,j2,j3,j4,(char)c1, (char)c2, (char)c3, (char)c4);
    }
```