```
/******************************************************************
 *
 *  Project.....:  isotropic FDTD code
 *  Application.:  generation of orientated mesh
 *  Module......:  mesh_base.cpp
 *  Description.:  Generate an oriented mesh on the surface of a cuboid
 *                 within the FDTD grid.
 *  Compiler....:  g++
 *  Written by..:  Peter Munro, Imperial College London, 2002-2008
 *  Environment.:  Linux
 *  Modified....:  Numerous times
 *
 ******************************************************************/

/*----------------------------------------------------------------*/
//                         INCLUDE section
/*----------------------------------------------------------------*/

#include "math.h"
#include <complex>
#include "matio.h"

using namespace std;
#include "matlabio.h"
#include "mesh_base.h"


/*Generate a matrix of vertices which define a triangulation of a regular
  two dimensional grid. This function assumes that the space of interest
  is a 2d surface with coordinates (i,j). I0 represents the lowest value
  of i for any point on the rectangular grid and I1 the highest. Similarly
  for j. A value of k is constant. A line of the output matrix looks like:

  i1 j1 k i2 j2 k i3 j3 k

  Triangles are taken by subdividing squares in the grid in a regular manner.

  coordmap is an integer array with three entries. This array can be a permutation of
  {0,1,2}. This defines the mapping between i,j,k and the indices in the output matrix.
  For example, if coordmap = {0,1,2} then a row in the matric would look like:

  i1 j1 k i2 j2 k i3 j3 k

  If, however, we have coordmap = {2,1,0} then we would get

  k j1 i1 k j2 i2 k j3 i3

  This should be interpreted as original i colums moves to column k. original k column moves to column i.


    i
 I1 ^ .  .  .  .
    | .  .  .  .
 I0 | .  .  .  .
    +------------>j
     J0        J1


  order specifies the direction of the surface normals of the triangles. This
  can take only 2 possible values +1 or -1. They have the following meaning:

  order = 1 means that the surface normal for a triangle in the:
                                xy plane will || to the z-axis
                                zy plane will || to the x-axis
                                xz plane will || to the negative z-axis
```

```
  order = -1 means surface normals are in the opposite direction. The surface normal
  is assumed to be in the direction (p2-p1)x(p3-p1) where p1-p3 are the points which
  define the triangle, in the order that they are listed in the facet matrix.

  The space allocated by *vertexMatrix must be freed after use.
*/

void triangulatePlane(int I0, int I1, int J0, int J1, int K,int coordmap[], int order, mxArray **vertexMatrix){
  int i, j, ndims, dims[2], counter = 0;
  int temp_res[] = {0,0,0};
  int **vertices;
  char buffer[100];

  //first some basic error checks
  /*  if( I1 <= I0 )
    mexErrMsgTxt("Error in triangulatePlane(), must have I1 > I0");
  if( J1 <= J0 )
    mexErrMsgTxt("Error in triangulatePlane(), must have J1 > J0");
  */
  //now check that coordmap is correct, should be a permutation on {0,1,2}
  for(i=0;i<=2;i++)
    for(j=0;j<=2;j++)
      temp_res[j] = temp_res[j] || coordmap[i]==j;

  //check all numbers are within range and none are equal
  if( !(temp_res[0] && temp_res[1] && temp_res[2]) || (coordmap[0]==coordmap[1]) || (coordmap[1]==coordmap[2]) || (coordmap[0]==coordmap[2])){
    sprintf(buffer,"Error in triangulatePlane(), coordmap incorrect [%d %d %d], [%d %d %d]",coordmap[0],coordmap[1],coordmap[2],temp_res[0],temp_res[1],temp_res[2]);
    mexErrMsgTxt(buffer);

  }

  ndims = 2;

  dims[1] = 9;                //each triangle has 3 vertices and each vertex has three indices
  dims[0] = 2*(I1-I0)*(J1-J0);//number of triangles


  *vertexMatrix =  mxCreateNumericArray( ndims, (const mwSize *)dims, mxINT32_CLASS, mxREAL);

  vertices = castMatlab2DArrayInt((int *)mxGetPr(*vertexMatrix), dims[0], dims[1]);

  if( !(order==1 || order==-1) )
    mexErrMsgTxt("Error in triangulatePlane(), order can take the value of +1 or -1");

  if( order == 1)
    for(j=J0;j<J1;j++)
      for(i=I0;i<I1;i++){
        //triangle 1
        //vertex 1
        vertices[coordmap[0]][counter]   =   i;
        vertices[coordmap[1]][counter]   =   j;
        vertices[coordmap[2]][counter]   =   K;

        ///vertex 2
        vertices[3+coordmap[0]][counter]   =   i+1;
        vertices[3+coordmap[1]][counter]   =   j;
        vertices[3+coordmap[2]][counter]   =   K;

        ///vertex 3
        vertices[6+coordmap[0]][counter]   =   i;
        vertices[6+coordmap[1]][counter]   =   j+1;
        vertices[6+coordmap[2]][counter++]   =   K;

        //triangle 2
        //vertex 1
        vertices[coordmap[0]][counter]   =   i+1;
        vertices[coordmap[1]][counter]   =   j;
        vertices[coordmap[2]][counter]   =   K;

        ///vertex 2
        vertices[6+coordmap[0]][counter]   =   i;
        vertices[6+coordmap[1]][counter]   =   j+1;
        vertices[6+coordmap[2]][counter]   =   K;

        ///vertex 3
        vertices[3+coordmap[0]][counter]   =   i+1;
        vertices[3+coordmap[1]][counter]   =   j+1;
        vertices[3+coordmap[2]][counter++]   =   K;

      }
  else
     for(j=J0;j<J1;j++)
      for(i=I0;i<I1;i++){
        //triangle 1
        //vertex 1
        vertices[coordmap[0]][counter]   =   i;
        vertices[coordmap[1]][counter]   =   j;
        vertices[coordmap[2]][counter]   =   K;

        ///vertex 2
        vertices[6+coordmap[0]][counter]   =   i+1;
        vertices[6+coordmap[1]][counter]   =   j;
        vertices[6+coordmap[2]][counter]   =   K;

        ///vertex 3
        vertices[3+coordmap[0]][counter]   =   i;
        vertices[3+coordmap[1]][counter]   =   j+1;
        vertices[3+coordmap[2]][counter++]   =   K;

        //triangle 2
        //vertex 1
        vertices[coordmap[0]][counter]   =   i+1;
        vertices[coordmap[1]][counter]   =   j;
        vertices[coordmap[2]][counter]   =   K;

        ///vertex 2
        vertices[3+coordmap[0]][counter]   =   i;
        vertices[3+coordmap[1]][counter]   =   j+1;
        vertices[3+coordmap[2]][counter]   =   K;

        ///vertex 3
        vertices[6+coordmap[0]][counter]   =   i+1;
        vertices[6+coordmap[1]][counter]   =   j+1;
        vertices[6+coordmap[2]][counter++]   =   K;

      }

  //now free memory

  freeCastMatlab2DArrayInt(vertices);
}

void trianglatePlaneSkip(int I0, int I1, int J0, int J1, int K,int coordmap[],
int order, mxArray **vertexMatrix, int dI, int dJ){
  int i, j, ndims, dims[2], counter = 0, countI = 0, countJ = 0;
  int temp_res[] = {0,0,0};
  int **vertices;
  char buffer[100];

  //first some basic error checks
```

```
  /*  if( I1 <= I0 )
    mexErrMsgTxt("Error in triangulatePlane(), must have I1 > I0");
  if( J1 <= J0 )
    mexErrMsgTxt("Error in triangulatePlane(), must have J1 > J0");
  */
  //now check that coordmap is correct, should be a permutation on {0,1,2}
  for(i=0;i<=2;i++)
    for(j=0;j<=2;j++)
      temp_res[j] = temp_res[j] || coordmap[i]==j;

  //check all numbers are within range and none are equal
  if( !(temp_res[0] && temp_res[1] && temp_res[2]) || (coordmap[0]==coordmap[1])
 || (coordmap[1]==coordmap[2]) || (coordmap[0]==coordmap[2])){
    sprintf(buffer,"Error in triangulatePlane(), coordmap incorrect [%d %d %d], [%d %d %d]",coordmap
[0],coordmap[1],coordmap[2],temp_res[0],temp_res[1],temp_res[2]);
    mexErrMsgTxt(buffer);

  }

  ndims = 2;

  dims[1] = 9;//each triangle has 3 vertices and each vertex has three indices

  for(i=I0;i<=I1;i=i+dI)
    countI++;
  for(j=J0;j<=J1;j=j+dJ)
    countJ++;


  dims[0] = 2*(countI-1)*(countJ-1);//number of triangles

  *vertexMatrix =  mxCreateNumericArray( ndims, (const mwSize *)dims, mxINT32_CL
ASS, mxREAL);
  vertices = castMatlab2DArrayInt((int *)mxGetPr(*vertexMatrix), dims[0], dims[1
]);
  if(1){
  if( !(order==1 || order==-1) )
    mexErrMsgTxt("Error in triangulatePlane(), order can take the value of +1 or -1");

  if( order == 1)
    for(j=J0;j<=(J1-dJ);j=j+dJ)
      for(i=I0;i<=(I1-dI);i=i+dI){
        //triangle 1
        //vertex 1
        vertices[coordmap[0]][counter]   =   i;
        vertices[coordmap[1]][counter]   =   j;
        vertices[coordmap[2]][counter]   =   K;

        ///vertex 2
        vertices[3+coordmap[0]][counter]   =   i+dI;
        vertices[3+coordmap[1]][counter]   =   j;
        vertices[3+coordmap[2]][counter]   =   K;

        ///vertex 3
        vertices[6+coordmap[0]][counter]   =   i;
        vertices[6+coordmap[1]][counter]   =   j+dJ;
        vertices[6+coordmap[2]][counter++]   =   K;

        //triangle 2
        //vertex 1
        vertices[coordmap[0]][counter]   =   i+dI;
        vertices[coordmap[1]][counter]   =   j;
        vertices[coordmap[2]][counter]   =   K;

        ///vertex 2
        vertices[6+coordmap[0]][counter]   =   i;
        vertices[6+coordmap[1]][counter]   =   j+dJ;
        vertices[6+coordmap[2]][counter]   =   K;
        ///vertex 3
        vertices[3+coordmap[0]][counter]   =   i+dI;
        vertices[3+coordmap[1]][counter]   =   j+dJ;
        vertices[3+coordmap[2]][counter++]   =   K;

      }
  else
    for(j=J0;j<=(J1-dJ);j=j+dJ)
      for(i=I0;i<=(I1-dI);i=i+dI){

        //triangle 1
        //vertex 1
        vertices[coordmap[0]][counter]   =   i;
        vertices[coordmap[1]][counter]   =   j;
        vertices[coordmap[2]][counter]   =   K;

        ///vertex 2
        vertices[6+coordmap[0]][counter]   =   i+dI;
        vertices[6+coordmap[1]][counter]   =   j;
        vertices[6+coordmap[2]][counter]   =   K;

        ///vertex 3
        vertices[3+coordmap[0]][counter]   =   i;
        vertices[3+coordmap[1]][counter]   =   j+dJ;
        vertices[3+coordmap[2]][counter++]   =   K;

        //triangle 2
        //vertex 1
        vertices[coordmap[0]][counter]   =   i+dI;
        vertices[coordmap[1]][counter]   =   j;
        vertices[coordmap[2]][counter]   =   K;

        ///vertex 2
        vertices[3+coordmap[0]][counter]   =   i;
        vertices[3+coordmap[1]][counter]   =   j+dJ;
        vertices[3+coordmap[2]][counter]   =   K;

        ///vertex 3
        vertices[6+coordmap[0]][counter]   =   i+dI;
        vertices[6+coordmap[1]][counter]   =   j+dJ;
        vertices[6+coordmap[2]][counter++]   =   K;

      }

  //now free memory

  freeCastMatlab2DArrayInt(vertices);
  }
}

/*vertexMatrix should be a 6 element array. Generates 6 arrays of facets using t
riangulatePlane.
 Each matrix is a plane of the cuboid which is defined by:

 (I0,I1)x(J0,J1)x(K0,K1)

 Each vertexMatrix[i] should be destroyed after calling this function
*/

void triangulateCuboid(int I0, int I1, int J0, int J1, int K0, int K1, mxArray *
*vertexMatrix){
  /*
  if( I1 <= I0 )
    mexErrMsgTxt("Error in triangulateCuboid(), must have I1 > I0");
  if( J1 <= J0 )
    mexErrMsgTxt("Error in triangulateCuboid(), must have J1 > J0");
  if( K1 <= K0 )
    mexErrMsgTxt("Error in triangulateCuboid(), must have K1 > K0");
  */
```

```
  int coordmap1[] = {0,1,2};
  int coordmap2[] = {1,2,0};
  int coordmap3[] = {0,2,1};

  triangulatePlane(I0, I1, J0, J1, K0, coordmap1, -1, &vertexMatrix[0]);//-ve z-
axis s norm
  triangulatePlane(I0, I1, J0, J1, K1, coordmap1,  1, &vertexMatrix[1]);//+ve z-
axis s norm

  triangulatePlane(J0, J1, K0, K1, I0, coordmap2, -1, &vertexMatrix[2]);//-ve x-
axis s norm
  triangulatePlane(J0, J1, K0, K1, I1, coordmap2,  1, &vertexMatrix[3]);//+ve x-
axis s norm

  triangulatePlane(I0, I1, K0, K1, J0, coordmap3,  1, &vertexMatrix[4]);//-ve y-
axis s norm
  triangulatePlane(I0, I1, K0, K1, J1, coordmap3, -1, &vertexMatrix[5]);//+ve y-
axis s norm
}

void triangulateCuboidSkip(int I0, int I1, int J0, int J1, int K0, int K1, mxArr
ay **vertexMatrix, int dI, int dJ, int dK){
  /*
  if( I1 <= I0 )
    mexErrMsgTxt("Error in triangulateCuboid(), must have I1 > I0");
  if( J1 <= J0 )
    mexErrMsgTxt("Error in triangulateCuboid(), must have J1 > J0");
  if( K1 <= K0 )
    mexErrMsgTxt("Error in triangulateCuboid(), must have K1 > K0");
  */
  int coordmap1[] = {0,1,2};
  int coordmap2[] = {1,2,0};
  int coordmap3[] = {0,2,1};
  triangulatePlaneSkip(I0, I1, J0, J1, K0, coordmap1, -1, &vertexMatrix[0], dI,
dJ);//-ve z-axis s norm
  triangulatePlaneSkip(I0, I1, J0, J1, K1, coordmap1,  1, &vertexMatrix[1], dI,
dJ);//+ve z-axis s norm

  triangulatePlaneSkip(J0, J1, K0, K1, I0, coordmap2, -1, &vertexMatrix[2], dJ,
dK);//-ve x-axis s norm
  triangulatePlaneSkip(J0, J1, K0, K1, I1, coordmap2,  1, &vertexMatrix[3], dJ,
dK);//+ve x-axis s norm

  triangulatePlaneSkip(I0, I1, K0, K1, J0, coordmap3,  1, &vertexMatrix[4], dI,
dK);//-ve y-axis s norm
  triangulatePlaneSkip(I0, I1, K0, K1, J1, coordmap3, -1, &vertexMatrix[5], dI,
dK);//+ve y-axis s norm
}


/* Generates a triangulation of a cuboid defined the surface of a regular
   grid. The result is returned in a concise manner, ie, a list of vertices
   and a list of facets which index in to the list of vertices.

   The list of vertices is itself a list of indices in to the x, y and z
   grid label vectors. In this sense this function deals only with the topology
   of the cuboid and the mesh. An extra step is required to generate the actual
   mesh from the values returned by this function.

   The surface of the volume [I0,I1]x[J0,J1]x[K0,K1] is meshed by
   this function.

   *vertices is an array of vertices, each row is a numbered vertex.

   *facets is an array of facets each of which is created using 3 vertex indices
.
    Each index is an index in to the vertices array.

*/
```

```
void conciseTriangulateCuboid(int I0, int I1, int J0, int J1, int K0, int K1,
                              mxArray **vertices, mxArray ** facets){



  mxArray *triangles[6];
  mxArray *index_map;
  int ndims, dims[3], ***index_map_int, nindices, **vertices_int, i, **facets_in
t, j,k, **triangles_int,ii,jj,kk;
  int ndims_v, dims_v[2];
  int ndims_f, dims_f[2];
  //int *dims_t;
  const mwSize *dims_t;
  int vertex_counter = 0, facets_counter = 0;
  int temp_vertex[3];

  /*
  if( I1 <= I0 )
    mexErrMsgTxt("Error in conciseTriangulateCuboid(), must have I1 > I0");
  if( J1 <= J0 )
    mexErrMsgTxt("Error in conciseTriangulateCuboid(), must have J1 > J0");
  if( K1 <= K0 )
    mexErrMsgTxt("Error in conciseTriangulateCuboid(), must have K1 > K0");
  */
  //this will keep count of the indices which have been allocated
  ndims = 3;
  dims[0] = I1-I0+1;
  dims[1] = J1-J0+1;
  dims[2] = K1-K0+1;

  index_map = mxCreateNumericArray( ndims, (const mwSize *)dims, mxINT32_CLASS,
mxREAL);
  index_map_int = castMatlab3DArrayInt((int *)mxGetPr(index_map), dims[0], dims[
1], dims[2]) ;

  //now initialise each entry to -1
  for(i=0;i<dims[0];i++)
    for(j=0;j<dims[1];j++)
      for(k=0;k<dims[2];k++)
        index_map_int[k][j][i] = -1;

  //the total number of indices that we will have
  nindices = (I1 - I0 + 1)*( J1 - J0 + 1)*2 + (I1 - I0 + 1)*(K1 - K0 - 1)*2 + (J
1 - J0 - 1)*(K1 - K0 - 1)*2;
  //fprintf(stderr,"%d [%d %d %d %d %d]\n",nindices,I0,I1,J0,J1,K0,K1);
  if( I1==I0 )
    nindices = (J1 - J0 + 1)*(K1 - K0 + 1);
  if( J1==J0 )
    nindices = (I1 - I0 + 1)*(K1 - K0 + 1);
  if( K1==K0 )
    nindices = (I1 - I0 + 1)*(J1 - J0 + 1);

  //construct vertice array
  ndims_v = 2;
  dims_v[0] = nindices;
  dims_v[1] = 3;
  *vertices = mxCreateNumericArray( ndims_v, (const mwSize *)dims_v, mxINT32_CLA
SS, mxREAL);
  vertices_int = castMatlab2DArrayInt((int *)mxGetPr(*vertices),dims_v[0], dims_
v[1]);
  //now generate triangles
  triangulateCuboid(I0,I1,J0,J1,K0,K1,triangles);

  //now setup the facet array
  ndims_f = 2;
  dims_f[0] = 4*(I1-I0)*(J1-J0) + 4*(J1-J0)*(K1-K0) + 4*(I1-I0)*(K1-K0) ;//the t
otal number of facets
  if( I1==I0 || J1==J0 || K1==K0 )
```

```
    dims_f[0] = dims_f[0]/2;
  dims_f[1] = 3;
  *facets = mxCreateNumericArray( ndims_f, (const mwSize *)dims_f, mxINT32_CLASS
, mxREAL);
  facets_int = castMatlab2DArrayInt((int *)mxGetPr(*facets), dims_f[0], dims_f[1
]);
  //now populate the matrices
  for(i=0;i<6;i++){//loop over each plane

    if( !(i==2 && I0==I1) && !(i==0 && K0==K1) && !(i==4 && J0==J1) ){

      dims_t = mxGetDimensions(triangles[i]);
      triangles_int = castMatlab2DArrayInt((int *)mxGetPr(triangles[i]), dims_t[
0], dims_t[1]);
      for(j=0;j<(int)dims_t[0];j++){//now iterate over triangle

        for(k=0;k<3;k++){//now each vertex in the triangle
          //first check if this vertex has been allocated
          kk = triangles_int[3*k+2][j];
          jj = triangles_int[3*k+1][j];
          ii = triangles_int[3*k][j];

          if( index_map_int[kk-K0][jj-J0][ii-I0] == -1){//not allocated yet
            index_map_int[kk-K0][jj-J0][ii-I0] = vertex_counter++;
            vertices_int[0][vertex_counter-1] = ii;
            vertices_int[1][vertex_counter-1] = jj;
            vertices_int[2][vertex_counter-1] = kk;

          }//of allocating new vertex
          temp_vertex[k] = index_map_int[kk-K0][jj-J0][ii-I0];

        }//of loop on each vertex
        facets_int[0][facets_counter] = temp_vertex[0];
        facets_int[1][facets_counter] = temp_vertex[1];
        facets_int[2][facets_counter++] = temp_vertex[2];

      }//of loope on each triangle
      freeCastMatlab2DArrayInt(triangles_int);
    }
  }//of loop over each plane


  //free memory etc
  freeCastMatlab3DArrayInt(index_map_int,dims[2]);
  freeCastMatlab2DArrayInt(facets_int);
  freeCastMatlab2DArrayInt(vertices_int);

  for(i=0;i<6;i++)
    mxDestroyArray(triangles[i]);
}

void conciseTriangulateCuboidSkip(int I0, int I1, int J0, int J1, int K0, int K1
,
                                  int dI, int dJ, int dK,
                                  mxArray **vertices, mxArray ** facets){

  mxArray *triangles[6];
  mxArray *index_map;
  int ndims, ***index_map_int, nindices, **vertices_int, i, **facets_int, j,k, *
*triangles_int,ii,jj,kk;
  int ndims_v;//, dims_v[2];
  int ndims_f;//, dims_f[2];
  //  int *dims_t;
  const mwSize *dims_t;
  mwSize *dims_v, *dims_f, *dims;
  int vertex_counter = 0, facets_counter = 0;
  int temp_vertex[3];

  dims_v = (mwSize *)malloc(2*sizeof(mwSize));
  dims_f = (mwSize *)malloc(2*sizeof(mwSize));

  /*
  if( I1 <= I0 )
    mexErrMsgTxt("Error in conciseTriangulateCuboid(), must have I1 > I0");
  if( J1 <= J0 )
    mexErrMsgTxt("Error in conciseTriangulateCuboid(), must have J1 > J0");
  if( K1 <= K0 )
    mexErrMsgTxt("Error in conciseTriangulateCuboid(), must have K1 > K0");
  */
  //this will keep count of the indices which have been allocated
  dims = (mwSize *)malloc(3*sizeof(mwSize));
  ndims = 3;
  dims[0] = (I1−I0)/dI+1;
  dims[1] = (J1−J0)/dJ+1;
  dims[2] = (K1−K0)/dK+1;

  index_map = mxCreateNumericArray( ndims, (const mwSize *)dims, mxINT32_CLASS,
mxREAL);
  index_map_int = castMatlab3DArrayInt((int *)mxGetPr(index_map), dims[0], dims[
1], dims[2]) ;

  //now initialise each entry to −1
  for(i=0;i<(int)dims[0];i++)
    for(j=0;j<(int)dims[1];j++)
      for(k=0;k<(int)dims[2];k++)
        index_map_int[k][j][i] = −1;
    //the total number of indices that we will have
  int Iw, Jw, Kw;
  Iw = (I1−I0)/dI+1;
  Jw = (J1−J0)/dJ+1;
  Kw = (K1−K0)/dK+1;

  nindices = Iw*Jw*2 + Iw*(Kw−2)*2 + (Jw−2)*(Kw−2)*2;
  //  fprintf(stderr,"%d [%d %d %d]\n",nindices,Iw,Jw,Kw);
  if( (I1−I0)<dI )
    nindices = Jw*Kw;
  if( (J1−J0)<dJ )
    nindices = Iw*Kw;
  if( (K1−K0)<dK )
    nindices = Iw*Jw;

  //construct vertice array
  ndims_v = 2;
  dims_v[0] = nindices;
  dims_v[1] = 3;
  *vertices = mxCreateNumericArray( ndims_v, (const mwSize *)dims_v, mxINT32_CLA
SS, mxREAL);

  vertices_int = castMatlab2DArrayInt((int *)mxGetPr(*vertices),dims_v[0], dims_
v[1]);
  //now generate triangles
  triangulateCuboidSkip(I0,I1,J0,J1,K0,K1,triangles,dI,dJ,dK);

  //now setup the facet array
  ndims_f = 2;
  dims_f[0] = 4*(Iw−1)*(Jw−1) + 4*(Jw−1)*(Kw−1) + 4*(Iw−1)*(Kw−1) ;//the total n
umber of facets
  if( (I1−I0)<dI || (J1−J0)<dJ || (K1−K0)<dK )
    dims_f[0] = dims_f[0]/2;
  dims_f[1] = 3;
  *facets = mxCreateNumericArray( ndims_f, (const mwSize *)dims_f, mxINT32_CLASS
, mxREAL);
  facets_int = castMatlab2DArrayInt((int *)mxGetPr(*facets), dims_f[0], dims_f[1
]);

  //now populate the matrices
  for(i=0;i<6;i++){//loop over each plane
    //    fprintf(stderr,"Here %d\n",i);
```

```
    if( !(i==2 && (I1−I0)<dI) && !(i==0 && (K1−K0)<dK) && !(i==4 && (J1−J0)<dJ)
){
      dims_t = mxGetDimensions(triangles[i]);
      triangles_int = castMatlab2DArrayInt((int *)mxGetPr(triangles[i]), dims_t[
0], dims_t[1]);

      for(j=0;j<(int)dims_t[0];j++){//now iterate over triangle

        for(k=0;k<3;k++){//now each vertex in the triangle
          //first check if this vertex has been allocated
          kk = triangles_int[3*k+2][j];
          jj = triangles_int[3*k+1][j];
          ii = triangles_int[3*k][j];
          if( index_map_int[(kk−K0)/dK][(jj−J0)/dJ][(ii−I0)/dI] == −1){//not all
ocated yet
            index_map_int[(kk−K0)/dK][(jj−J0)/dJ][(ii−I0)/dI] = vertex_counter++
;
            vertices_int[0][vertex_counter−1] = ii;
            vertices_int[1][vertex_counter−1] = jj;
            vertices_int[2][vertex_counter−1] = kk;

          }//of allocating new vertex
          temp_vertex[k] = index_map_int[(kk−K0)/dK][(jj−J0)/dJ][(ii−I0)/dI];
        }//of loop on each vertex
        facets_int[0][facets_counter] = temp_vertex[0];
        facets_int[1][facets_counter] = temp_vertex[1];
        facets_int[2][facets_counter++] = temp_vertex[2];
      }//of loope on each triangle
      freeCastMatlab2DArrayInt(triangles_int);
    }
  }//of loop over each plane

  //free memory etc
  freeCastMatlab3DArrayInt(index_map_int,dims[2]);
  freeCastMatlab2DArrayInt(facets_int);
  freeCastMatlab2DArrayInt(vertices_int);

  for(i=0;i<6;i++)
    mxDestroyArray(triangles[i]);
  free(dims);
  free(dims_v);
  free(dims_f);
}


/*Determines the vector which points from p1 to p2*/
void pointsToVector(int p1[], int p2[], int *vector){
  int i;
  for(i=0;i<3;i++)
    vector[i] = p2[i] − p1[i];

}

/*Calculates v1xv2*/
void crossProduct(int v1[], int v2[], int *v1crossv2){
  v1crossv2[0] = v1[1]*v2[2] − v1[2]*v2[1];
  v1crossv2[1] = v1[2]*v2[0] − v1[0]*v2[2];
  v1crossv2[2] = v1[0]*v2[1] − v1[1]*v2[0];
}


/*
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,const mxArray *prhs[]){
  int I0, I1, J0, J1, K,K0,K1, dI, dJ,dK, counter = 0,coordmap[3];//triangulate
plane
  int *p1, *p2, *p3, v1[3], v2[3], *cross, dims[2];//testing cross product

  int coordmap1[] = {0,1,2};
  if(nrhs != 9)
      mexErrMsgTxt("Incorrect number of input parameters");

    I0 = ((int) *mxGetPr(prhs[counter++]));
    I1 = ((int) *mxGetPr(prhs[counter++]));
    J0 = ((int) *mxGetPr(prhs[counter++]));
    J1 = ((int) *mxGetPr(prhs[counter++]));

    //
    //K = ((int) *mxGetPr(prhs[counter++]));

    K0 = ((int) *mxGetPr(prhs[counter++]));
    K1 = ((int) *mxGetPr(prhs[counter++]));
    dI = ((int) *mxGetPr(prhs[counter++]));
    dJ = ((int) *mxGetPr(prhs[counter++]));
    dK = ((int) *mxGetPr(prhs[counter++]));

    if( nlhs != 4)
      mexErrMsgTxt("Must have 6 output argument");


    conciseTriangulateCuboidSkip(I0, I1, J0, J1, K0, K1,
                                 dI, dJ, dK,
                                 (mxArray **)plhs, (mxArray **)(plhs+1));
    conciseTriangulateCuboid(I0, I1, J0, J1, K0, K1,
                          (mxArray **)(plhs+2), (mxArray **)(plhs+3));
    //triangulatePlaneSkip(I0, I1, J0, J1, K,coordmap1, 1, (mxArray **)&plhs[0],
 dI, dJ);
    //triangulatePlane(I0, I1, J0, J1, K,coordmap1, 1, (mxArray **)&plhs[1]);
    //triangulateCuboidSkip(I0, I1, J0, J1, K0, K1,(mxArray **)plhs,dI, dJ,dK);
    //triangulateCuboid(I0, I1, J0, J1, K0, K1,(mxArray **)(plhs+6));
    //[o11 o12 o13 o14 o15 o16 o21 o22 o23 o24 o25 o26] = mesh_base(1,3,1,3,1,3,
2,2,2)
    //[o11 o12 o13 o14 o15 o16] = mesh_base(1,3,1,3,1,3,2,2,2)
    //mex −v mesh_base.cpp /home/ptpc2/prmunro/code/ptws1/matlablibrary/matlabio
3/matlabio.cpp −I/home/ptpc2/prmunro/code/ptws1/matlablibrary/matlabio/

    //conciseTriangulateCuboid(I0, I1, J0, J1, K0, K1, (mxArray **)&plhs[0], (mx
Array **)&plhs[1]);



}
*/
```