JAN 25 1998

~~Review & Approval Desk MS 0619 (1)~~

~~For DOE/OSTI~~

SANDIA REPORT

SAND98-2717

Unlimited Release

Printed December 1998

RECEIVED

FEB 09 1999

dgtoexo2: A Distorted Grid Output File to Exodus II Finite Element Database Conversion Utility

Harry K. Moffat

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

SAND98-2717
Unlimited Release
Printed December 1998

dgtoexo2: A Distorted Grid Output File to Exodus II Finite Element Database Conversion Utility*

Harry K. Moffat
Chemical Processing Sciences Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0601

Abstract

This report describes how to obtain publication-quality graphics from distorted grid electronic structure codes using the combination of the conversion utility, `dgtoexo2`, and `mustafa`, an AVS Express application. `dgtoexo2` converts scalar function results from a format applicable to distorted grid codes into the Exodus II unstructured finite element data representation. `mustafa` can read Exodus II files and use the AVS Express engine to visualize data on unix and Windows NT platforms. Though not designed for the purpose, the `dgtoexo2/Exodus II/mustafa` combination is sufficiently versatile to provide for the specialized graphics needs of electronic structure codes. The combination also scales well, producing robust performance for problems involving millions of grid points.

*This work was supported by the U. S. Department of Energy, Sandia National Laboratories under Contract DE-AC04-94AL85000. There exists a non-exclusive license for use of this work by or on behalf of the U. S. Government.

Acknowledgment

The author thanks Mike Glass for many helpful discussions concerning the mustafa graphics package.

Contents

Acknowledgment	4
Introduction.....	7
Description of the Translator	8
Distorted Grid Representation	8
Description of the Finite Element Representation	11
Running the Programs.....	13
Running dgtoexo2.....	13
Running mustafa	14
Identifying Crystal Planes: slicer_plane	15
Tips for Obtaining Publishable Graphics from mustafa	17
Future Additions to the Translator	20
References.....	21

Figures

Figure 1. Structure representing the distorted grid data.....	9
Figure 2. Show_Mesh Wire Frame Visualization with SPH Particle Field Visualization used to view atom positions for a 2x2x2 unit cell.	10
Figure 3. Structure for the finite element representation of the data.....	11
Figure 4. Subset of the structure representing information defined for atoms in the FE representation - the rest of the structure is unused.....	12
Figure 5. Show_Mesh Hidden Line visualization with SPH Particle Field Visualization. No red atoms are visible; they are inside the unit cell.	13
Figure 6. A combination of External_Edges visualization, Show_Mesh visualization with hidden lines, and SPH Particle Field visualization	14
Figure 7. Example of slicer_plane visualization technique combined with the SPH Particle Field and External Edge Visualization techniques. Some of the atoms are in the slicer plane and some are not.....	16
Figure 8. Example of the isovolume visualization technique. Regions of high charge density are surrounded by a volume. The values of rho are painted onto the outside of the volume.....	19

Intentionally Left Blank

Introduction

This report describes how to obtain publication-quality graphics from distorted grid electronic structure codes using the combination of the conversion utility, `dgtoexo2`, and `mustafa`, an AVS Express application. `dgtoexo2` converts scalar function results from distorted grid codes into the Exodus II unstructured finite element data representation [3]. `dgtoexo2` is cognizant of translational symmetry and can take advantage of it. Unit cells in directions of translational symmetry are completed when translating into the Exodus II data representation. Also, multiple unit cells may be generated in translational symmetry directions to provide the viewer with a clearer picture of the symmetry in the problem.

`mustafa` is an official AVS Express application written by Mike Glass, Div. 9111. A reduced price license can be bought from AVS to run the `mustafa` application without having to pay for the full AVS Express product. `mustafa` reads Exodus files and then uses the AVS Express engine to visualize the data. It provides a Motif GUI interface to the visualization methods most pertinent to fluid mechanics and structural mechanics applications. `mustafa`'s capabilities are also sufficient for the visualization of data from electronic structure codes.

Distorted grid codes employ topologically rectilinear grids. Being an unstructured finite element format, Exodus II can handle much more complicated meshes than rectilinear ones. Thus, the conversion to Exodus II has made the underlying data structure more complex than the original calculation in order to visualize it. Is this necessary? The short answer is that `dgtoexo2/mustafa` is sufficiently robust, and it was the easiest approach given the computational infrastructure at Sandia. The question that I had posed above about whether there is a penalty in going to an unstructured representation of originally structured data is really one of computational speed. The `mustafa`/AVS Express combination has shown itself in practice to be sufficiently fast that speed is not a concern. Also, the ability to customize a front end to `mustafa`, i.e., the translator, `dgtoexo2`, meant that the specialized graphics visualization needs could be accommodated with `mustafa`'s existing visualization methods.

The computational infrastructure at Sandia favored `mustafa`. Two unix graphics packages were looked into to provide a graphics solution: IDL and AVS Express. IDL did not have any canned solutions for visualizing volumetric data on non-uniform grids. One attempted solution was to translate the distorted grid solutions onto a regular grid. IDL then had a canned solution, `slicer3D`, to visualize volumetric information from regular grids. However, the C program to translate data on a distorted grid to a regular grid turned out to be slow. Also, the eventual graphics results didn't turn out to be very pleasing. At least my initial graphics attempts didn't turn out very well, and there was no local IDL expert to go to for advice. Modules to visualize the mesh would also have required custom subroutines written in IDL's parsing language that would have operated on a very low level (writing out each line in the mesh). There were several positive points to using IDL. It's cheap on a per platform basis; there is a large IDL user base and internet repositories of third party IDL graphics functions. Also, the company behind the IDL product, RSI, takes its customer technical support function very seriously. I would heartily recommend IDL for applications in which it is traditionally very strong: signal processing, 2D image processing, and 1D plotting where a cheaply constructed GUI interface is desired.

The bare AVS or AVS Express product may have direct support for topologically rectilinear grids. However, there is no existing infrastructure that I could work from. I would have had to create my own networks from scratch. Additionally, I would have had to figure out a way to provide standardized input to AVS, probably by writing a translation program equivalent to the one I've written for the Exodus II solution, anyway. Instead, I felt that it was better to go with a translator to Exodus II and take advantage of the large user base here at Sandia. Once the distorted grid results are in Exodus II format, several graphics packages are available to view the files [1, 4]. The one most suitable for 3D viewing is `mustafa`, which is an application that sits on top of AVS Express, built with the

Express application development environment. Additionally, AVS visualization techniques using self-built networks is also an unexplored but attractive possibility. Mike Glass, mustafa's author, has written a `read_exoII()` module for AVS that may be added to networks. A benefit of translation to the Exodus II database format is that Exodus II files are written in machine independent binary [3]. Exodus II is written on top of UCAR's netCDF package [2], which uses the ubiquitous XDR package for its low level IO routines.

I will first describe how `dgtoexo2` is structured. Then, I will briefly describe how to run the `dgtoexo2/mustafa` combination to obtain visualization results of interest to electronic structure applications. `slicer_plane`, a utility program for the exact specification of visualization planes, will also be presented. An informal discussion of how to obtain publication-quality printed output will also be provided, because the potential for wasting time by an unwary user in obtaining this bottom-line result is high. Finally, a to-do list for directions in which the `dgtoexo2` may be improved is provided. All figures in this report come from a single SiC calculation carried out by Alan Wright and Ellen Stechel on a $30 \times 30 \times 30$ grid with translational symmetry assumed in all three directions.

Description of the Translator

`dgtoexo2` is a mixed Ansi C and Fortran 77 code. Ansi C is used on the top levels of the program to provide a structural representation of the data and to provide dynamic memory allocation. Fortran 77 is used to parse ascii input files, because of the author's familiarity with the Chemkin Fortran 77 library. Parsing subroutines from that library, which are robust and have been heavily tested, have been used in this application. A full set of Ansi C prototypes have been created for these Fortran 77 parsing routines to ensure that the interface between programming languages is handled correctly. Ansi C has been used in the subroutine that writes the Exodus II data file, again because of the authors familiarity with that particular interface.

The internal operation of `dgtoexo2` can be described by a few operations involving the top level structures. The first operation of reading the description and data from the distorted grid codes is described in the next section. The intermediate operations of translating the distorted grid code data from a distorted grid data structure to a finite element data structure is described in the section entitled, Description of the Finite Element Representation. The final operation of writing the data out to the Exodus II file is also described in that section.

Distorted Grid Representation

This section will describe how `dgtoexo2` reads the results from a distorted grid code. Currently, it reads the output from Alan Wright's `mkccs` program. `mkccs` is a translator program that combines the results from several fortran binary files into one ascii file. `dgtoexo2` reads the `mkccs` output file via a fairly flexible, sophisticated parsing algorithm, that allows for a variable number of scalar functions to be defined on the distorted grid. However, there are some formatting limitations in the header file format of the `mkccs` ascii output file that must be adhered to when reading that file. The reader is referred to the `mkccs` source code for a description of the formatting requirements of the header. `dgtoexo2` also expects to find and read a local file named `latfile`. `latfile` contains information about the lattice vectors in the problem. Rudimentary checking is carried out between the ascii output file from `mkccs` and the file, `latfile`, in order to make sure that two files are compatible. However, this process is clearly not bullet-proof.

Figure 1 describes the `DG_Struct` structure, which is the end result of this rather ill-defined input process. `Num_x`, `Num_y`, and `Num_z` are variables that describe the number of grid points in the three primary directions. `NumFuncValues` is equal to the number of scalar variables defined on the mesh. The actual value of the function variables are stored in

FuncValues[k][j], where k is the k^{th} scalar function and j is the j^{th} node. FuncNames[k] is the character string name of the k^{th} scalar function, also read in from mkccs's output file. The name of each scalar variable is taken from the line, "function: name1, nam2...", that precedes the actual columnar listing of the data. DistortedGridSpace is a flag to indicate what types of coordinates are to be used in the structure. The default is to use the real space coordinates, i.e., those read in from the mkccs input file. Distorted grid codes employ topologically rectilinear structured meshes. The grid point locations are not regularly spaced in real space. Node points are clustered around regions of large gradients in the charge density in order to generate better accuracy. If DistortedGridSpace is true, the real-space coordinates, *Coords[3] in the structure, are overwritten with mesh coordinates generated from a uniform meshing algorithm based on an equidistribution of points on the parallelepiped domain. This is the called "Distorted Grid Space", because it is the coordinate system in which the solution actually takes place.

```

struct DG_Struct {
    int Num_x;           /* Number of nodes in the x direction */
    int Num_y;           /* Number of nodes in the y direction */
    int Num_z;           /* Number of nodes in the z direction */
    int NNodes;          /* Total number of nodes in the structure */
    int NumFuncValues;    /* Number of scalar functions defined on the
                           distorted grid */
    int NumRecords; /* Read in from the ascii file, but otherwise ignored */
    int DistortedGridSpace;
        /* Flag to indicate whether the coordinates in Coords are in
        the distorted grid space or not. If they are, then the nodal
        coordinates will be generated by uniformly gridding along
        the lattice vector directions. If they are not (which is
        the default), then the nodal coordinates will be obtained
        from the distorted grid output file. */
    int NumAtoms;        /* Number of Atoms */
    int NumTypes;        /* Number of different types of atoms */
    int TranslationalSymmetry[3]; /* Logical flag indicating the presense
                                   of translational symmetry in each
                                   direction */
    double LatticeVectors[3][3];
        /* Translational periodicity in the given direction
        LatticeVectors[i][0]: This is the component of
        the i_th lattice vector in the x direction. */
    double *Coords[3]; /* (x,y,z) Coordinates for the nodes. This
                        is an array of three pointers to doubles.
                        Each array has a length NNodes. */
    double **FuncValues; /* Values of the scalar functions defined on the
                        distorted grid mesh:
                        FuncValues[j][i] is the value of the jth function
                        at the ith mesh point. */
    char **FuncNames; /* Vector of character strings representing the
                        names of the functions */
    double AtomPositions[MAXATOMS][3]; /* (x,y,z) locations of the atoms */
    int AtomID[MAXATOMS]; /* Type of the atom -> this number must exist as
                           an entry in the TypeID vector. It usually
                           ranges from 1 up to the total number of different
                           types of atoms */
        /* Information about each type of atom */
    int TypeID[MAXTYPES]; /* Type ID for the type */
    int TypeAtomNum[MAXTYPES]; /* TypeAtomNum[i] is the atomic number of i */
    double TypeStrength[MAXTYPES];
    double TypeClength[MAXTYPES];
    double TypeCoeff[MAXTYPES];
    char Description[80]; /* Textual description of the current data */
};

```

Figure 1. Structure representing the distorted grid data

The TranslationalSymmetry[3] and LatticeVectors[3][3] variables have to do with the existence of translational symmetry and its specification within the problem statement. Translational symmetry is specified by lattice vectors. Lattice vectors have the

Intentionally Left Blank

property that translation via integer units of a lattice vector yields identical scalar function values. If there is symmetry in all directions, then there will be three lattice vectors, and the concept of a unit cell may be used. The unit cell is specified by a parallelepiped determined by vectors emanating from a single point with lengths and directions equal to the three lattice vectors. The mesh constructed by distorted grid codes span this unit cell. The last mesh plane in each direction of the unit cell is typically dropped in the calculation and in writing out the solution to the output file. Translational symmetry requires the last mesh plane in the unit cell to be equal to the first mesh plane. It is important when visualizing the solutions to add this last plane back into the mesh to make evident the translational symmetry in the problem. This process of adding the last plane back in I have nicknamed, "completing" the unit cell. By default, completing the unit cell is done when translating the distorted grid results represented by Fig. 1 into the finite element representation. There is an internal flag in `dgtoexo2` to turn off the completion of the last unit cell, if so desired.

Figure 2. Show_Mesh Wire Frame Visualization with SPH Particle Field visualization used to view atom positions for a 2x2x2 unit cell. The last unit cell in each lattice vector direction is completed, generating more atoms of type 1, blue, than there are of type 2, red, since the blue atom was located at the origin of the unit cell while the red atom is located inside the unit cell.

Atom positions and properties of atom types are also stored in the `DG_Struct` structure. The positions and type ID for the i^{th} atom in the cell are stored in `AtomPosition[i][3]` and `AtomID[i]`, respectively. `AtomicNumber[j]`, `TypeStrength[j]`, `TypeClength[j]`, `TypeCoeff[j]`, and `TypeID[j]` refer to properties associated with the j^{th} type of atom. `TypeID[j]` is the type ID of the j^{th} type of atom. A

Intentionally Left Blank

lookup function, `findTypePos()`, is used to associate type ID's with the index position of types of atoms, using the array, `TypeID[]`.

In the future, input modules based on the reading the direct binary file output from a distorted grid code may be constructed. That's the strength of the current structured, modular approach. The essential point is that the `DG_Struct` structure of Figure 1 must be fully populated, so that the conversion to a finite element data structure described in the next section, may proceed without knowledge of the origin of the data.

```

struct FE_Mesh_Struct {
    char    *Title;           /* Title of the problem */
    int     Num_x;            /* Number of nodes in the x direction */
    int     Num_y;            /* Number of nodes in the y direction */
    int     Num_z;            /* Number of nodes in the z direction */
    int     NElemTotal;       /* Total Number of elements in the structure */
    int     NNodes;           /* Total number of nodes in the structure */
    double  *Coords[3];       /* Coordinates for the nodes. This is an array
                               of three pointers to doubles. Each array
                               has a length of NNodes. */
    int     NumElemBlks;      /* Number of element blocks in the mesh */
    int     *NElemBlk;        /* Number of elements in the element block */
    char    **ElemType;       /* Element type for elements in the block */
    int     *ElemBlkIDs;      /* Element block id for the element blocks */
    int     *NumAttrPerElem;  /* Number of attributes per element in the element
                               block */
    int     *NumNodesPerElem; /* Number of nodes per element */
    int     **Connect;        /* Connectivity matrix. Connect[i] is the
                               connectivity vector for element block, i */
    double  **ElemAttr;       /* Vector of element attributes
                               ElemAttr[k] is a pointer to the vector of attributes
                               fomr the kth element block */
    int     DistortedGridSpace;
    /* Flag to indicate whether the coordinates in Coords are in
       the distorted grid space or not. If they are, then the nodal
       coordinates will be generated by uniformly gridding along
       the lattice vector directions. If they are not (which is
       the default), then the nodal coordinates will be obtained
       from the distorted grid output file. */
    int     NumAtoms;         /* Number of atoms */
    int     NumTypes;         /* Number of different types of atoms */
    ATOM_STRUCT *Atoms;       /* Vector of pointers to atom structures
                               Atoms[i] is the atom structure for the ith
                               atom in finite element description. */
    int     NumNodeSets;      /* Number of node sets defined in the mesh */
    struct  NodeSetStruct *NodeSetData; /* Data for each node set */
    int     NumSideSets;      /* Number of side sets defined in the problem */
    struct  SideSetStruct *SideSetData; /* Data for each side set */
    int     NumFuncValues;    /* Number of scalar functions defined on nodes */
    double  **FuncValues;     /* Value of the functions at the nodes
                               FuncValues[i] is a vector containing
                               the value of the ith function at all nodes */
    char    **FuncNames;      /* Vector of strings containing the
                               of the functions, FuncNames[i] is the name
                               of the ith function */
}

```

Figure 3. Structure for the finite element representation of the data

Description of the Finite Element Representation

The translation process from the distorted grid representation to the finite element model representation is carried out in two steps. In the first step, the mesh representation and data on the mesh are translated. In the second step, the atom representation described in Fig. 1 is added onto finite element model using additional nodes and spherical elements defined at new nodes. Both steps are complicated by completing the last unit cell and by creating multiplications of the unit cell in directions of translational symmetry. Fig. 2 is an example. For a 2x2x2 unit cell representation, where the first type of atom is located at the

origin and the second type of atom is located in the middle of the unit cell, 27 atoms of the first type and 8 atoms of the second type are generated.

Figure 3 contains the top level structure which defines how the finite element data is internally represented in `dgtoexo2`. This representation is closely related to the Exodus II application program interface (API). Therefore, the code to write out the data to an Exodus II file is short and straightforward. The first step, the translation of the mesh representation, is carried out by treating each meshed region as a hexagonal finite element. Trilinear interpolation is used in the hexagon and in `mustafa`. Thus, the first type of element in the list of types of elements, `ElemType[0]`, will be of type "HEX8". A connectivity array for the mesh in the first element block, `Connect[0]`, is built. The connectivity array is a one dimensional vector of length `NElemBlk[0]*NumNodesPerElem[0]`, which describes the node-to-element mapping of the mesh [3].

Entries for node sets and side sets in the `FE_Mesh_Struct` structure are not used in this application. `FuncValues[k][i]` contains the values of all variables defined at nodes. `k` refers to the element block, while `i` refers to the node number. There are no element based variables defined in the `FE_Mesh_Struct` structure. The `DistortedGridSpace` flag is propagated from the distorted grid structure to indicate what type of coordinates, `Coords[3][i]`, have been propagated from that data structure. The finite element mesh data structure doesn't include any of the entries describing the symmetry in the problem. Symmetry properties have already been applied in forming the elements of the `FE_Mesh_Struct` structure, so their description is no longer needed.

```
struct Atom_Struct {
    double   Position[3];
    double   TypeStrength;
    double   TypeClength;
    double   TypeCoeff;
    int      AtomicNumber; /* Atomic number of the atom */
    char     *Name;
    int      Type;          /* Type of the atom -> this number must range
                           from 1 up to the total number of different
                           types of atoms */
    double   Radius_for_Vis;
};
```

Figure 4. Subset of the structure representing information defined for atoms in the FE representation - the rest of the structure is unused.

In the second step, atom positions and atom type information is added to the `FE_Mesh_Struct` structure. This is done by adding additional element blocks of spherical elements for each different type of atom. Each spherical element is connected to one underlying node through the connectivity matrix, `Connect[k][j]`. The position of the sphere is specified by the position of the node. New nodes have been added to the overall list of nodes for the sole purpose of specifying atomic positions. This is done for two reasons. The first is that atom positions are not restricted to falling on nodes of the underlying distorted grid geometry. The other reason is that the addition of new nodes creates new nodal variables, which can then be used to advantage within `mustafa`, as described below. Each spherical element takes one attribute, the radius of the sphere. An initial radius of 1.0 is inserted into the `ElemAttr[k][j]` array (`k` is the element block number, and `j` is the element number in the k^{th} block). The relative sizes of all atoms can be modified later within `mustafa`. The new version of `mustafa` supports the visualization of spherical elements, specified by a position and radius, via the SPH Particle Field visualization method. In that method, atoms are colored according to the value of a particular nodal variable. I have included logic in `dgtoexo2` to encode the type of the atom into one of the nodal variables, by default the variable entitled "J^{1/2}". Thus, different types of atoms will be visualized within `mustafa` using different colors.

Atom information is also included in `FE_Mesh_Struct` in a duplicate way. A vector of pointers to `Atom_Struct` structures of length, `NumAtoms`, is created. Figure 4 contains

the definition of the Atom_Struct structure. It encompasses all of the atom type information originally contained in the distorted grid structure.

Running the Programs

Running dgtoexo2

The translator program is invoked via the following command line.

```
dgtoexo2 [-m multiUnitCell] [-dgs] dgName [exoName]
```

Errors in command line usage will generate a printout of the correct usage on standard output. dgName, the only required parameter, is the name of the output file from mkccs. The -dgs token is optional, and if specified, changes the coordinates in the finite element Exodus II file to distorted grid space coordinates. The -m multiUnitCell option specifies the number of unit cells in each lattice vector direction, multiUnitCell is equal to the number of cells requested. The ExodusII output file name, exoName, is optional, because default names are generated. If the -dgs option is specified, the default Exodus II file name is the value of dgName with the suffix, "_dgs.exoII" appended to it. If the -dgs option is not used, the Exodus II file name is the value of dgName with the suffix, ".exoII" appended to it. Additionally, a small ascii file named "latfile" must be present in the local directory. Examples of the format are given in the dgtoexo2 distribution directory.

Figure 5. Show_Mesh Hidden Line visualization with SPH Particle Field visualization. Red atoms inside the unit cell are invisible. Mesh is two pixels wide.

Intentionally Left Blank

Running mustafa

`mustafa` operates via a fairly simple paradigm at its top-most level. First, the Exodus II mesh is read in using the File -> Exodus II/PDS (PXI)... menu command. Then, various filters can be used to cull the objects in the Exodus II file, modify the nodal positions, or mirror the Exodus II model about a plane of symmetry. I have found these filters to be unnecessary for the electronic structure visualizations. Then, various visualization methods are used to display the model and data in a view window. There can be more than one view window. Usually, I just use one view window at a time. Within a view window, more than one visualization method can be visible at any time. For example, many of the images created in this report have been generated by additive combinations of visualization methods. Each visualization method has a group of widget controls associated with it. Those controls are visible in `mustafa`'s widget control window when that particular visualization method has been "selected." Visualization methods can also either be visible or invisible. Being invisible means that `mustafa` still remembers the method and the current value of its widget controls; however, it is just not visible in the current view window..

Figure 6. A combination of `External_Edges` visualization, `Show_Mesh` visualization with hidden lines, and `SPH Particle Field` visualization

Intentionally Left Blank

Other useful visualization methods include the isovolume and isosurface methods. The two methods are very similar in that they both explore what surfaces of a constant scalar function look like in three dimensions. Both methods have the capability to create a surface using one variable and then color that surface using another variable. The isovolume method, example of which is provided in Fig. 8, renders a volume surrounding two isosurfaces and the element block boundaries. The isosurface visualization method provides a rudimentary capability for creating traditional contour plots within *mustafa*.

Identifying Crystal Planes: *slicer_plane*

One mismatch of the visualization needs of electronic structure codes with *mustafa*'s capabilities has been remedied with an auxiliary program. For electronic structure calculations, one would desire representations of scalar quantities on planes whose normal and translational position are exactly specified. However, AVS Express only provides widget controls meant to provide fuzzy positioning of slicer planes. Therefore, there needs to be an additional utility that can exactly position slicer planes in *mustafa* according to the value of the surface normal. I wrote a stand-alone program, *slicer_plane*, to accomplish this task.

slicer_plane will calculate the values of the widget controls that must be entered into the plane transform dialog box of *mustafa* slicer plane visualization method. *slicer_plane* takes as its input, three points. These three points define the visualization plane. More than a generic specification of a plane is actually needed for the slicer widget control. The slicer widget is a finite-dimensioned rectangle with an origin, a vector that defines the right-hand-side of the plane and another vector perpendicular to the right-hand-side vector that defines the left side of the rectangle. The extent of the rectangle can be further specified via a scale input widget control box that increases or decreases the relative size of its initial extent. The initial extent is determined roughly by the data being plotted.

slicer_plane will use the first point to specify the origin of the slice plane. It will use the difference between the second point and the origin of the first point to specify the right-hand-side of the slicer plane used by *mustafa*. The third data point is differenced with respect to the first data point to create the second vector needed to uniquely identify the surface normal. With these additional definitions, the orientation and position of the slicer widget is completely specified. It's still not unique, however, because there is a degeneracy with respect to the top and bottom surface normal still to account for.

The algorithm used in *slicer_plane* starts with the method for finding the right-side vector of the slicer plane. The right side of the slicer plane transforms especially easily. With all three widget controls set at zero, the right side vector of the slicer plane falls along the positive x axis, and the surface normal points in the positive z direction. The x angle widget control doesn't affect the orientation of the right-side vector; it does affect the value of the surface normal.

$$\begin{bmatrix} \cos \theta_z \cos -\theta_y \\ -\sin \theta_z \cos -\theta_y \\ \sin(-\theta_y) \end{bmatrix} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} = \mathbf{r} \quad (1)$$

$\mathbf{r}$ is the normalized vector representing the right-side vector of the plane. θ_y and θ_z are the values of the angle widget controls in the plane transform dialog box. There are always two solutions to the above equation. They correspond to the slicer plane being flipped 180 degrees about the right side line, i.e., the top and bottom surface normals respectively. The degeneracy is related by the following transformation:

Intentionally Left Blank

$$\begin{aligned}(\theta_z' &= \theta_z + 180^\circ) \\ (\theta_y' &= 180^\circ - \theta_y)\end{aligned}\tag{2}$$

θ_y' and θ_z' are also valid solutions to Eqn. 1. Thus, we restrict our initial solution search for $-\theta_y$ to specify that $-\theta_y$ always be in the +x quadrants, i.e., $-\pi/2 \leq -\theta_y \leq \pi/2$. With this restriction, the solution to Eqn. 1 becomes unique and straightforward.

Figure 7. Example of slicer_plane visualization technique combined with the SPH Particle Field and External Edge visualization techniques. Some of the atoms are in the slicer plane and some are not. The origin and axes are also made visible by turning on that option in the view window of *mustafa*. The external edge visualization method provides a means to draw a bounding box about the computational domain.

The slicer plane in AVS Express is oriented via all three angle widgets. The normal to the slicer plane may be calculated by a formula involving the following rotational matrices operating on the z axis vector, which is the initial orientation of the slicer plane's surface normal:

$$\text{z rotational matrix: } \begin{bmatrix} \cos\theta_z & \sin\theta_z & 0 \\ -\sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix} = ROTZM$$

Intentionally Left Blank

$$\text{y rotational matrix: } \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix} = ROTYM$$

$$\text{x rotational matrix: } \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix} = ROTXM$$

$$[N] = [ROTZM][ROTYM][ROTXM] \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad (3)$$

N is the vector specifying the surface normal of the slicer plane. We have already determined θ_y and θ_z from the calculation of the right-side vector. Thus, we only need Eqn. 3 to calculate θ_x , and we have completely specified the three angle widgets.

An example of how to run `slicer_plane` in conjunction with `mustafa`'s slicer plane visualization technique is provided below. Let's say that we want to obtain the charge density along a plane that cuts diagonally through the center of the unit cell containing the atom in the middle of the unit cell. If the three lattice vectors of the unit cell are (4.09, 0, 4.09), (0, 4.09, 4.09), and (4.09, 4.09, 0), then such a plane is specified by setting the right-side vector to the first lattice vector, and the other vector to the sum of the remaining two lattice vectors. A `slicer_plane` sessions would then produce the following output:

```
zuzax.pchem.sandia.gov 218: slicer_plane
Enter three points on three separate lines:
0. 0. 0.
4.09 0. 4.09
4.09 8.18 4.09
NORMAL = (-.707106781186548,.0,.7071067811865475)
Set Widget controls to:
  Turn on Absolute toggle switch
  Theta_X = .000
  Theta = -45.000
  Thetas = .000
  Translate X = .000
  Translate Y = .000
  Translate Z = .000
  Adjust the Scale dialog box to encompass the data
```

These widget control settings on the plane transform box were used to produce the orientation of the slicer plane in Figure 7.

Tips for Obtaining Publishable Graphics from `mustafa`

All `mustafa` results have been created with version 1.24.3, July 24, 1998. The underlying version number of AVS Express was v. 3.4. The current version number of the Exodus library application is v. 2.06. All of the above influence the tips that I give below. Bugs come and go with evolving version numbers of these applications and with the base system graphics software. In general, it is a good idea to keep up to date with vendor-supplied bug fixes for the graphics subsystem and rendering software on your computer.

To obtain quality output from `mustafa`, the best technique is to use the tiff export option from Output->Write_Raster_File dialog box. The reason for this is that the images

eventually all end up as bitmaps anyway. Even the regular postscript output option produces a bitmap. Tiff images have the advantage that they may be created with a known bits per inch resolution. If care is taken never to scale the image by a non-integer factor, direct mapping between the bits in the image and the bits in the output device can be made to occur, leading to the best possible output. Tiff files exported from *mustafa* may be imported as images into Framemaker and other programs. In Framemaker, the resolution of the image in bits-per-inch may be specified at the time the image is imported. This serves to scale the image using integer factors. A value of 600 bits per inch ensures that value of resolution in the final printed image. All of the images created in this report were generated using this procedure. There is a known bug in AVS Express with scaling xfont text in the tiff export method. These fonts don't scale with the resolution of the image in the tiff output file. However, for quality output, it has been suggested to not include text in the export image but to add it later in Framemaker or some other graphics program such as Adobe Illustrator. Note, stroke text, the other font type option for legends does scale, but doesn't look particularly good to start with. An example of stroke text is given in Fig. 6. and in Fig. 7.

There are several options in the Output->Write_Raster_file dialog box. The most important is the image scale option. The image scale determines the ratio of the number of pixels created in the tiff file to the number of pixels actually on the screen. For example, a 4 to 1 scale creates a tiff images with 16 times more pixels than the current screen size. Most importantly, tiff files with extra pixels have better resolution. Intelligent choices are made for the new pixels. Lines which are defined to be one pixel wide at screen resolution are one pixel wide at the scaled image resolution. This can create extremely narrow lines, when the image scale is set to 4 to 1 (see Fig. 2 for an example of this). The default RLE (run length encoding) compression option was used in all cases. Other types of compression algorithms didn't work within *mustafa* on HP-UX 10.20 (a bug report was generated). Even with the compression, 4-to-1 tiff files are typically 15 megabytes in size and take 15 minutes each to print on networked laser printers. The result, as mentioned above, is that the images have true 600 bits-per-inch resolution. The final image size can also be related to the original *Mustafa* window size by simple arithmetic. For example, if the final image size is 6 inches by 5 inches and 4 to 1 scaling and 600 dots per inch resolution is required, then the initial *Mustafa* window size should be set to 900 by 750 bits. The end product is spectacular. All of the images in this report were written into Framemaker at 600 dpi density. They were printed on a Tektronics Phaser 550 laserprinter with 600 dpi resolution. I see no reason why the tiff output file procedure can't be carried out at even higher resolutions such as 1200 dpi resolution, the minimum resolution needed for high quality images used in magazine covers.

There are two types of postscript output from *mustafa*, regular and vector postscript. The regular postscript option puts out a bitmapped image. Therefore, it has all of the pitfalls of using bit-mapped images and none of the benefits of using the tiff image format mentioned above. It too suffers from the xfont scaling problem. To increase the resolution of bit-mapped postscript images, the "3D horizontal resolution" slider bar in the Write Postscript/CGM dialog box can be increased.

The vector postscript output produced unpleasing output due to a triangularization of some of the images. This is a known bug in AVS Express whose time scale for resolution is unknown. The toggle box option, color subdivision, with its associated dialog box entitled color tolerance may help to reduce the effect of these triangles. The default option is to use the color subdivision scheme, "None". Changing the option to "Precompute" and then fiddling with the color tolerance value may help the triangle problem (decreasing the color tolerance value certainly leads to monumentally large output files quickly). The fonts in vector postscript come out looking great, however. I have recently discovered that the spherical primitives render as points in vector postscript output, no matter what rendering scheme was used. A bug report was generated.

Tiff files created by *mustafa* are readily importable into MS Word and Adobe Illustrator on the Mac and PC. However, memory limitations prevented me from reading in large tiff files into these applications on my Mac.

Different renderers do produce different results. The choice of renderers can be made in the Editors->View dialog box in *mustafa*'s view window. The software renderer option is available on all platforms. The default renderer on HP/UX systems, PEX, has a bug in it. PEX will display spherical primitives (such as the ones used for displaying spherical elements in *mustafa*) as points instead of as finite spheres. On HP systems, change the default renderer to OpenGL or to the software renderer in order to display the atoms correctly using the SPH Particle Field visualization method.

Under the Editor->Camera dialog box of the view window is another important widget control. The camera jiggle control determines how much accuracy is used in the hidden lines algorithm within *mustafa*. Frequently the default value of 1.0 isn't sufficient for proper rendering of large meshes created with *dgtoexo2*. Setting this widget control to 0.1 solves these rendering issues.

Figure 8. Example of the isovolume visualization technique. Regions of high charge density are surrounded by a volume. The values of ρ are painted onto the outside of the volume. Textual annotations are also added.

The default background color in *mustafa* is black. To change this to white, go to the Editors->View dialog box in the View window of *mustafa*. The default background color is specified there. I have found white to be the best background color for printing, and it doesn't waste toner. Using a white background, however, requires you to change the color settings on the fonts and mesh lines from the default white to black in order for these

Intentionally Left Blank

to be visible. In output dialog boxes where the background mode needs to be specified, I invariably use the mode, "keep". This produces no unexpected results.

Typically, one would like to print slicer_plane images in 2D mode. Once one has gone through the trouble of defining a plane and all of the image content is in that plane, one would like to make sure that that plane's normal is orthogonal to the picture screen and thus to the eventual printed page. This objective can be achieved using two steps, assuming that the plane's surface normal is given by $(\theta_x, \theta_y, \theta_z)$ values which are round numbers. The first step is to adjust the image of the plane so that its surface normal is roughly orthogonal to the screen. Then, go into the dialog box, Editors->Transform, and adjust the angle widget controls there to round numbers. Chances are that the objective of a 2D picture will have been achieved.

Future Additions to the Translator, dgtoexo2

Below is a to-do list. The single biggest improvement to the robustness in the code is one that would improve the interface between the distorted grid code and dgtoexo2. A more precise definition of the output file format from the distorted grid codes would help tremendously. The best solution would involve the adoption of netCDF [2] for output file format of the distorted grid code. However, barring this, adding the information contained in "latfile" to the the mkccs output file would be a step in the right direction, since the problem statement would reside with the problem data.

The visualization of a scalar function on one grid plane of the distorted grid may be a desirable feature. Because each grid plane is curved in space, the slicer plane visualization technique would not be sufficient to accomplish this task. However, using the concept of side sets [3], visualization of a grid plane could be achieved. I've included hooks into the FE_Mesh_Struct structure, specifically the pointer, struct SideSetStruct *SideSetData, to accommodate the definition of side sets into the finite element representation of the grid. An identified grid plane could then be defined as a side set in Exodus II lingo and written out to the Exodus II data base. A capability which will be included in mustafa at some point if it isn't already is to visualize data on side sets by using colors to represent the value of a function. Thus, a curved isosurface with colors representing scalar function data defined on the surface may be visualized.

Another approach to visualizing functions on surfaces created by the distorted grid is to add three nodal-based variables to the Exodus file. These nodal based variables would correspond to the original grid integer grid offsets (i.e., i, j, k nomenclature). Then, one could use these new variables in the Scalar->Iso-surface visualization method to plot variables on curved surfaces of constant grid coordinate (e.g., surfaces of constant i). The iso-surface tool has the capability to form the surface using one variable but create the color map on that surface according to the value of another variable.

I had originally planned to visualize atom positions by visualizing spheres meshed with hexagonal elements. It would not take too long to implement this approach, if there were a need to visualize information centered at atomic positions.

dgtoexo2 has only been tested on problems with translational symmetry in all three lattice directions. At a low level it has been designed to be capable of handling cases with less symmetry. However, with any software program, a feature which has not been tested may not work. dgtoexo2 should also be capable of generating meshes with different multiples of the unit cell in each lattice vector direction. Again, this capability hasn't been tested.

The use of trilinear interpolation between each grid point may not adequately reflect the true quality of the data produced by the distorted grid electronic structure code. These codes use a sophisticated global coordinate transformation of the unit cell based on a smooth Jacobian that is anything but trilinear. Also, the scalar functions calculated from the electronic structure code, which employ spectral techniques, use a global basis set

representation. A local finite element representation may not be functionally adequate. Thus, interpolating scalar variables within `mustafa` based on a trilinear approximation may produce inaccurate values. How inaccurate are the interpolations? Does accuracy really matter in a graphics visualization application? One approach to quantify these questions would be to bisect all grid points in the domains. Then, calculate the values of the scalar functions at the new grid points using a method consistent with the original basis set representation and coordinate transformation. An error tolerance can be introduced at this point to make decisions about whether the addition of the new grid points is warranted or whether additional bisections should occur. `mustafa`, as well as the underlying graphics rendering software, is based on a localized interpolation scheme. This can't be changed. What can be changed is adding resolution to what's fed into `mustafa`.

To enhance `slicer_plane` visualization plots, the capability to only include atoms in the finite element database that lie along a particular plane should be added. A potential source of confusion when looking at plots created with the `slicer plane` visualization method in combination with the SPH Particle Field visualization method would be eradicated. Currently, it is hard to distinguish whether atoms actually lie in the `slicer plane` or not.

The `dgtoexo2/mustafa` combination scales well to millions of nodal points. For example, I created a 1.7 million node finite element model using `dgtoexo2` in 30 seconds and easily manipulated it within `mustafa` on an HP J2240 workstation with an FX-2 graphics board. Thus, considering its versatility and its scalability, The `dgtoexo2/mustafa` combination may have appeal, after appropriate modification, for the visualization needs of other large scale density functional calculations done at Sandia.

References

1. M. W. Glass, "MUSTAFA User's Guide Version 1.23.x," Sandia Report, SAND#, Sandia National Laboratories, Albuquerque, NM 1997.
2. R. K. Rew, G. P. Davis, S. Emmerson, H. Davies, "NetCDF User's Guide - An Interface for Data Access, Version 2.4," University Corporation for Atmospheric Research, Boulder, Colorado, Feb. 1996.
3. L. A. Schoof, V. R. Yarberry, "EXODUS II: A Finite Element Data Model," Sandia Report, SAND92-2137, Sandia National Laboratories, Albuquerque, NM 1994.
4. A. P. Gilkey, J. H. Glick, "BLOT - A Mesh and Curve Plot Program for the Output of a Finite Element Analysis," Sandia Report, SAND88-1432, Sandia National Laboratories, Albuquerque, NM 1992.

DISTRIBUTION

5	MS0601	1126	H. K. Moffat
1	MS0601	1126	J. Y. Tsao
1	MS0601	1113	J. S. Nelson
1	MS0826	9111	M. K. Glass
1	MS0826	9111	W. L. Hermina
1	MS1111	9221	J. S. Shadid
1	MS1111	9221	A. S. Salinger
3	MS1415	1113	A. F. Wright
1	MS1417	1100	T. S. Picraux
1	MS1421	1153	N. Modine
2	MS0899	4916	Technical Library
1	MS9018	8940-2	Central Technical Files
1	MS0619	15102	Review & Approval Desk for DOE/OSTI