

The Jmol Voxel (JVXL) File Format

Robert M. Hanson, Department of Chemistry, St. Olaf College, Northfield, Minnesota, USA
55057, hansonr@stolaf.edu, <http://www.stolaf.edu/people/hansonr>
May 25, 2006

(revised 3/21/2007)

see <http://www.stolaf.edu/academics/chemapps/jmol/docs/examples-11/isosurface.htm>

Abstract

A format specification for the Jmol voxel (JVXL) file format is proposed. The purpose of the JVXL file format is to provide a mechanism for the efficient delivery of molecular surface data (orbitals, electron density plots, electrostatic potential maps, etc.) from a web server to a client page in a compact manner. The format was designed to be used specifically with the open-source Jmol molecular viewing and analysis applet (<http://jmol.sourceforge.net>), but the format has general utility anywhere a Gaussian CUBE file might be employed. Whereas typical CUBE files are on the order of 1-10 Mb in size, the derived JVXL files are 400-1000 times smaller, on the order of 5-10Kb. The Jmol applet can read and write the JVXL file format, and currently it is the only application that can be used to create JVXL files.

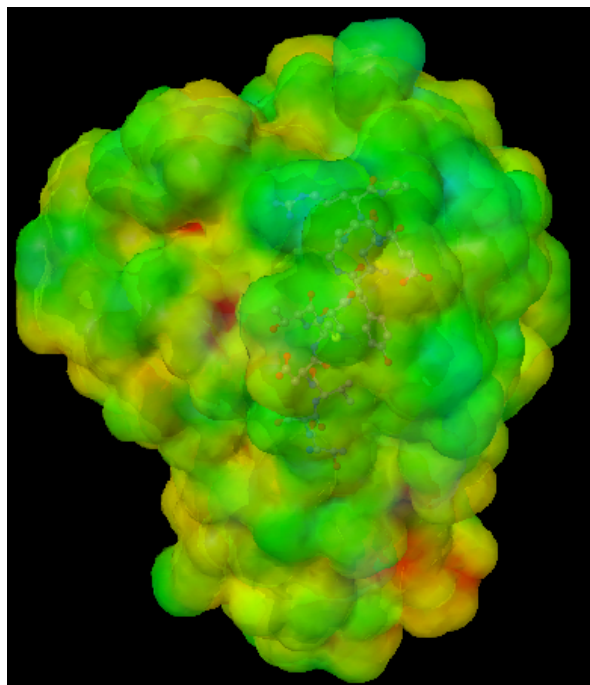
Background

Computed molecular orbitals and electron density data are typically stored in Gaussian CUBE file format [<http://local.wasp.uwa.edu.au/~pbourke/dataformats/cube/>]. Electrostatic potential map data for biomolecules can be produced by the open source APBS (Adaptive Poisson-Boltzmann Solver) program [<http://apbs.sourceforge.net/>], which saves the data in a “multigrid DX” format similar to Gaussian’s CUBE format. These files, even for small molecules, tend to be inconveniently large in the context of the web – on the order of 1-10 Mb even for relatively small molecules such as acrolein¹ and benzene² and much larger for proteins. The core data section of a CUBE file consists of a list of numerical values, each representing the value of some particular molecular quantity at a specific x,y,z position within a molecular space. The number of data points involved depends upon the resolution of the grid and the size of the space. It is not uncommon for CUBE files to hold from 10,000 to 500,000 data points, each requiring 13 bytes of storage space. CUBE files may contain any number of such lists. The size of cube files has proven a formidable deterrent to the real-time display of three-dimensional models of molecular surfaces on the web.

The JVXL Solution

Whereas the CUBE file contains the complete data set associated with a molecular parameter, typical use of CUBE file data is to display only a two-dimensional *surface* – the 95% probability surface for a molecular orbital, or the electrostatic potential of a molecule mapped onto an electron density surface. It is this characteristic use of the CUBE file that makes the JVXL format of great utility, for the JVXL file contains all the information needed to depict a finite, predetermined number of surfaces *derived* from the data stored in one or more CUBE files. This information turns out to be representable in far fewer bytes than the complete data set. The trade-off is that JVXL files do not contain the entire data set – they cannot be used to regenerate CUBE files, and they cannot be used to display any more surfaces than the designer has loaded

them with in the first place. Nonetheless, since the typical use of CUBE files is to present one or possibly a small number of surface representations, this is not in general a problem.

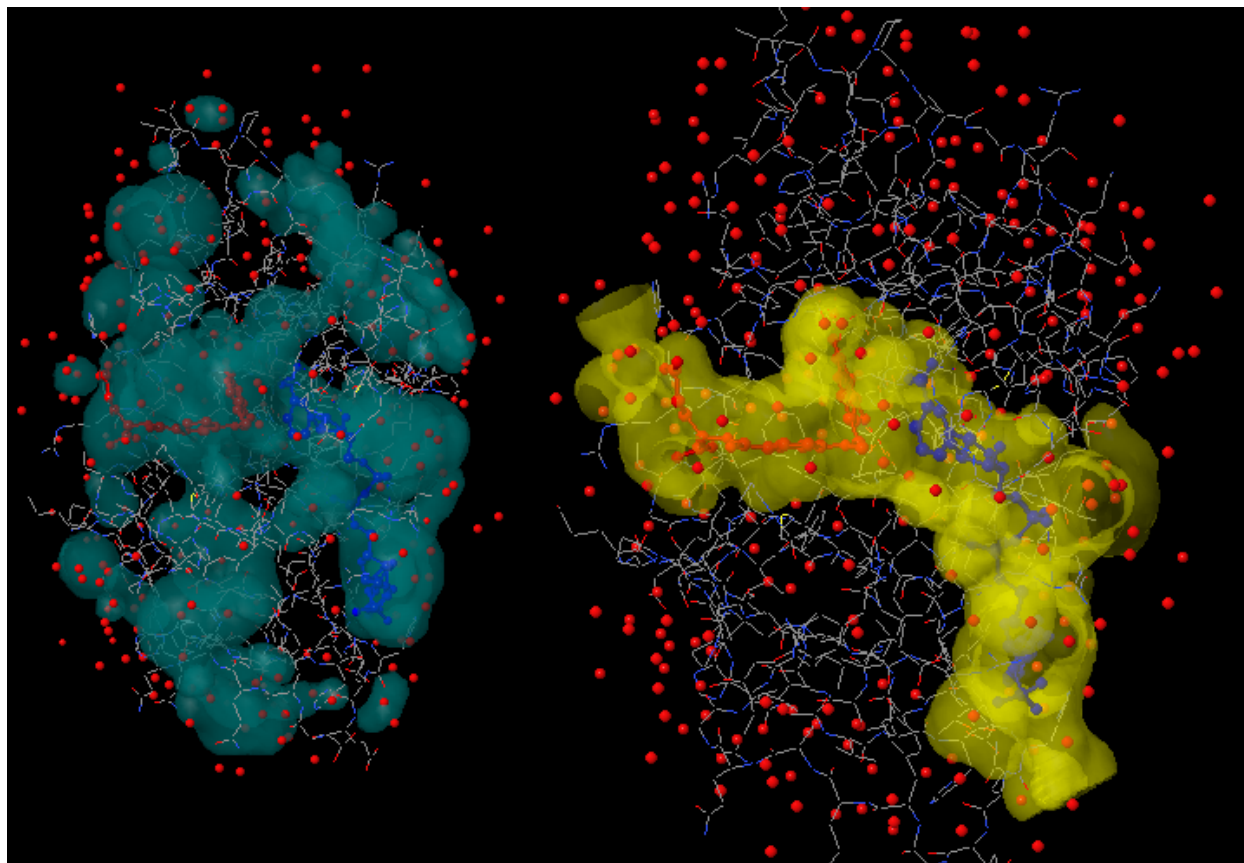


LEFT:

A map of the electrostatic potential of 1dry.pdb onto the van der Waals-radius surface.
(65K JVXL file 1dry-mep.jvxl)

BELOW:

Cavities in 3dfr.pdb rendered using Jmol; file sizes 27K (left), and 37K (right).



General JVXL File Formats

The JVXL file consists of two sections, a *header* section, and a *data* section, consisting of one or more *surface descriptions*. The header section is identical to the CUBE file header section, though freer in format. (Indeed, Jmol reads JVXL files with the same reader it uses for CUBE files in terms of reading atom coordinates.) The header section consists of the following subsets:

TABLE 1. JVXL (CUBE) FILE HEADER SECTION					
Lines 1 and 2: comments	Title Card Required potential=scf Electrostatic potential from Total SCF Density				
Line 3: a negative number (-N) representing the number of atoms followed by the origin of the “voxel space”, (x,y,z). <i>N must be negative; it cannot be zero</i> . Units are Bohr unless the optional keyword ANGSTROMS (with or without the brackets) is also on that line.	-5	-8.140940	-8.140940	-8.643459	[ANGSTROMS]
	-N	x	y	z	
Lines 4: the number of data points along the “X” coordinate (NX), as defined by the vector (x,y,z).	50	0.333333	0.000000	0.000000	
	NX	x	y	z	
Line 5: same for Y	50	0.000000	0.333333	0.000000	
Line 6: same for Z	55	0.000000	0.000000	0.333333	
Lines (6+1) – (6+N): Atomic number in integer and real format along with Cartesian coordinates of each of the N atoms. <i>There must be at least one atom</i> .	6	6.000000	0.000000	0.000000	-2.130707
	1	1.000000	0.000000	1.932284	-2.775380
	1	1.000000	1.673407	-0.966142	-2.775380
	1	1.000000	-1.673407	-0.966142	-2.775380
	17	17.000000	0.000000	0.000000	1.241787
Line 7 + N: A negative number (-NS) indicating the number of surface specifications, followed by four encoding numbers	-1 35 90 35 90 Jmol voxel format version 0.9b -NS EB ER CB CR				

(EB, ER, CB, CR) described below, and any additional text, possibly indicating the file format version	
--	--

It is not important that all atoms in a molecular system be represented here. The only significant difference between the JVXL and CUBE headers is that for the JVXL file at least one atom must be indicated, because a negative number for the number of atoms in a cube file indicates that the (7+N)th header line will be present. The negative number on that (7+N)th line distinguishes this file from a Gaussian CUBE file, where a *positive* number indicates the number of data set lists that follow.

In a CUBE file what now follows is an (NX x NY x NZ)-long list of numbers. In a JVXL file what follows is a data section, consisting of (NS) surface descriptions. The four numbers EB, ER, CB, and CR indicate how edge and color data were encoded into ASCII character format. Currently, these numbers are fixed by the Jmol software and are not user-adjustable.

Surface descriptions are of several types:

- Unmapped curved isosurfaces (Table 2).
- Color-Mapped isosurfaces (Table 3).
- Color-Contoured planar slices through CUBE data (Table 4).

Unmapped curved isosurfaces (Table 2). These surfaces are either derived from CUBE data by specifying a cutoff value or derived from the model itself in the form of a “solvent excluded surface” or “solvent-accessible surface”. In Jmol the script command is simply:

```
isosurface myname 0.05 "myfile.cube"
```

or

```
set solvent 1.4; isosurface solvent
```

or using a JVXL file, where the cutoff has been predetermined:

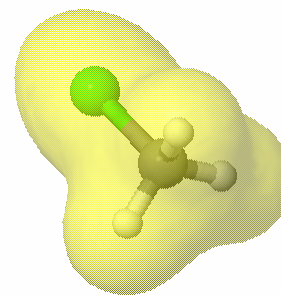
```
isosurface myname "myfile.jvxl"
```

Color-Mapped isosurfaces (Table 3). These are isosurfaces that have been overlaid with data from a second CUBE file. The Jmol script command allows for the optional designation of a range over which the colors red to blue should be applied:

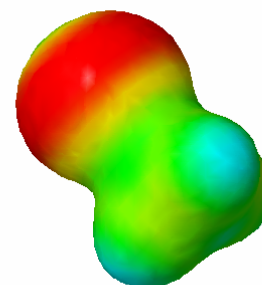
```
isosurface myname 0.05 "myfile.cube"  
color absolute 0.0 0.2 "mydata.cube"
```

The JVXL equivalent requires nothing more than for an unmapped isosurface, since a JVXL file includes the cutoff and coloring ranges as predetermined values:

```
isosurface myname "myfile.jvxl"
```



Jmol

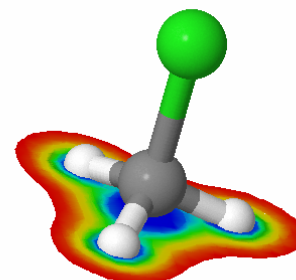


Jmol

The JVXL format also allows for a “high precision” color compression that takes up about twice as much space but allows for remapping of the colors by the end user. See the note under **high precision color**, below.

Color-Contoured planar slices through CUBE data (Table 4).

These are color-contoured surfaces created by first generating the intersection points of a plane with a voxel grid, then using those points as a basis for a marching squares (2D) contouring of the planar grid of data.



Jmol

TABLE 2. JVXL FILE SURFACE SPECIFICATION (nonplanar surface)	
Variable number of blank lines or comments	# comments begin with a # sign; blank lines are ignored # any number of such lines are allowed
Surface description line: the cutoff used for the computation of the surface (CO, for information only), the number of bytes of grid point data (NP), the number of bytes of edge data (NE), and a flag (NC) indicating whether colors data are present (NC=NE) or not (NC=-1) anything else on this line is informational only	0.02 6457 8076 -1 compressionRatio=462.06357 CO NP NE NC
One or more lines of integer <i>surface voxel bitmap data</i> (described below)	115922 2 6333 4 91 6 90 7 88 8 88 8 89 6 91 . . .
One or more lines of ASCII-encoded <i>edge fraction data</i> (described below)	3_+I6B3qPV4LVSwC{/K'+G]fcUy6Il<3; . . .
One or more lines of ASCII-encoded <i>color mapping data</i> (optionally compressed; described below)	7777889989:;;::<==:<=###\$#####\$\$%22223333###\$. . .

TABLE 3. JVXL FILE SURFACE SPECIFICATION (planar surface)	
Variable number of blank lines or comments	# comments begin with a # sign; blank lines are ignored # any number of such lines are allowed
Surface description line with plane indicated by -1 in both NP and NE. Four additional parameters (PA, PB, PC, and PD) defining the plane are required.	0.0 -1 -1 2500 0.0 0.0 1.0 0.0 CO NP NE NC PA PB PC PD
One or more lines of ASCII-encoded and compressed <i>color mapping data</i> .	#~871 \$~6 #~40 \$\$\$\$&'''&\$\$#~38 \$%')+, ,+) . . .

JVXL Surface Data

To understand what is present in the three surface data fields, one must understand the essentials of the *isosurface* method. In this method, the space surrounding a surface is considered to be a set of points laid out in a rectilinear or parallelepiped grid. (The format does not require that the axes be orthogonal, but typically they are.) An isosurface is defined as a surface through these points where a given parameter has a constant value (Figure 1).

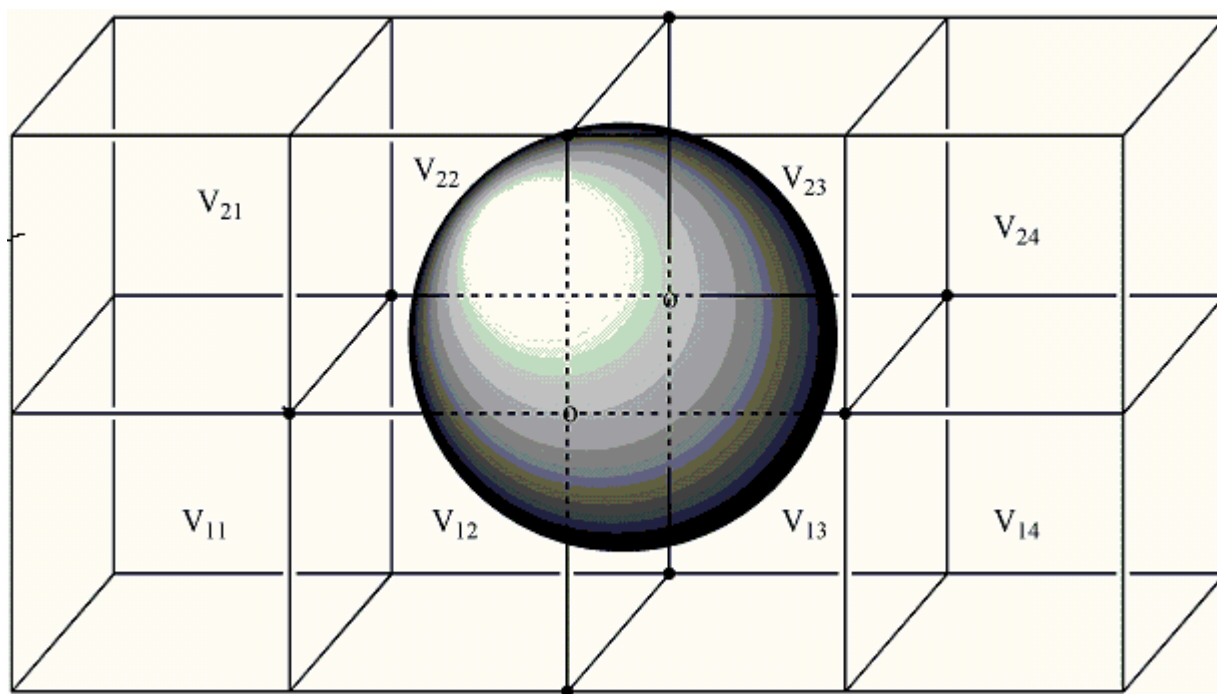


Figure 1. An isosurface surrounded by a 5 x 2 x 3 grid of data points. The key point is that much of the data points are unnecessary.

The first thing to understand is that the only significant information we require are the approximate positions where the surface cuts the lines between the data points. It will be these intersection points that are then turned into a series of triangles for surface rendering. Thus, no information in volumes V_{11} , V_{21} , V_{14} , or V_{24} are required. In fact, what we need to do is focus on the *edges*. We need only consider the edges where one end is on one side of the surface and the other end is on the other side – edges cut by the surface. If we can identify which edges these are and where along each edge the surface cuts, we have all we need. Figure 2 shows a close-up of one such surface.

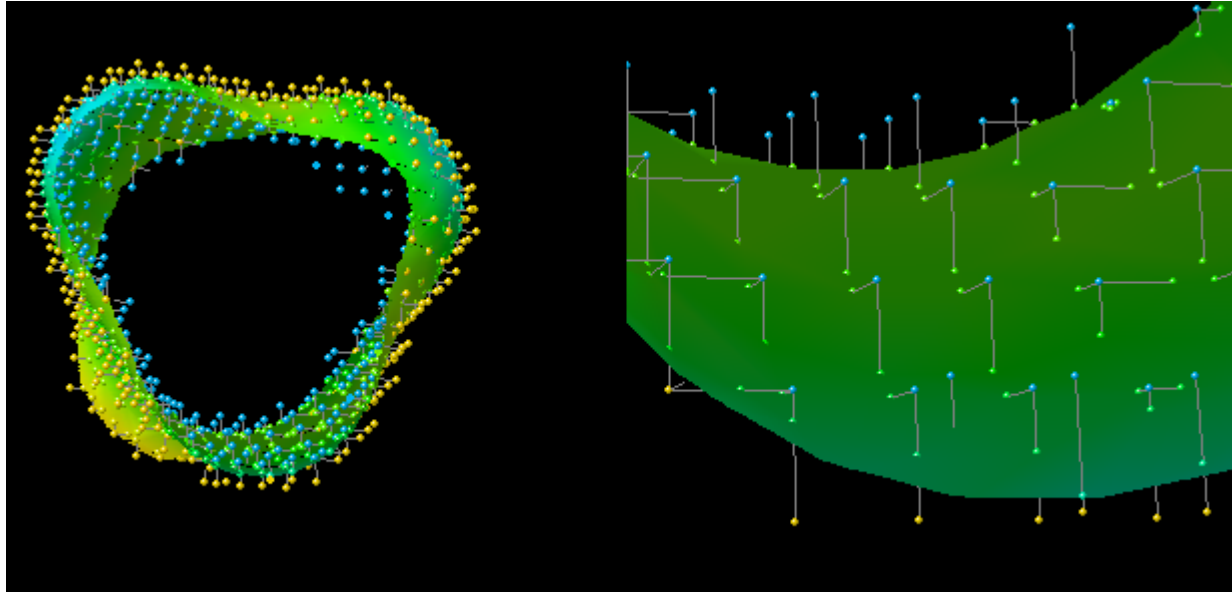


Figure 2. Two slices through a surface showing the relationship between the surface points used for triangulation (green) at intersections of the surface with edges of the grid.

The first set of surface data in a JVXL file maps out which data points are critical for surface construction – only those that terminate intersecting edges. Some are “inside”, and some are “outside” the surface. We simply count along in a systematic way. Using the for loop:

```
for (int x = voxelCountX; --x >= 0;) {
    for (int y = voxelCountY; --y >= 0;) {
        for (int z = voxelCountZ; --z >= 0;) {
```

we run through all the “voxels” and determine whether the value at each is closer to 0 than a predetermined cutoff value (“outside”) or further from 0 (“inside”). We simply list the number of voxels found sequentially on each side: 115922 outside, 2 inside, 6333 outside, 4 inside... forming a relatively compact yet readable *surface voxel bitmap*.

Edge Fraction Data

The surface voxel bitmap data is enough information to be able to reconstruct which edges are the “critical” edges. Having done that, we go back through the voxels using the *marching cube* algorithm,³ this time running through each critical edge and estimating the position of the intersection of the surface with the edge. The edges are numbered and gone through sequentially from highest numbered to lowest (11 to 0, Figure 3) using a clever method that never checks any edge twice.

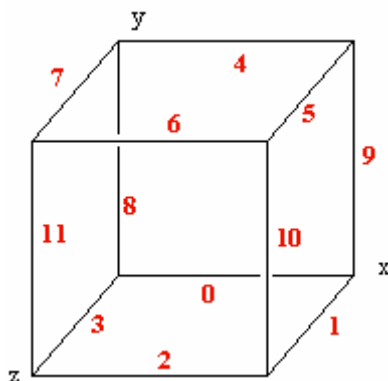


Figure 3. Edge numbering.

Only edges that have one end inside and one end outside are recorded. Based on the two values for the property at each end of the edge and the known cutoff value, a fraction is determined by linear interpolation:

$$\text{fraction} = (\text{cutoff} - \text{valueA}) / (\text{valueB} - \text{valueA});$$

It is this set of critical-edge fractions, then, that is encoded in the form of the *edge fraction data*. Encoding is carried out simply by determining the ASCII character that is that fraction of the way from the edge base character (EB, 35, '#') to an end character (125, '}') along a range of ASCII values (ER, 90). Since the fraction is rounded *down* to the nearest integer, the actual range is from 0 to ER – 1. The end character itself is not represented in this series. (A point at the “end” of one edge would be at the “beginning” of another and is not recorded twice.) The end character itself (125, '}') is reserved for “no value,” thus allowing for surface fragments. (In Jmol the value at that position in the grid is recorded as “Not-A-Number”, Float.NaN.)

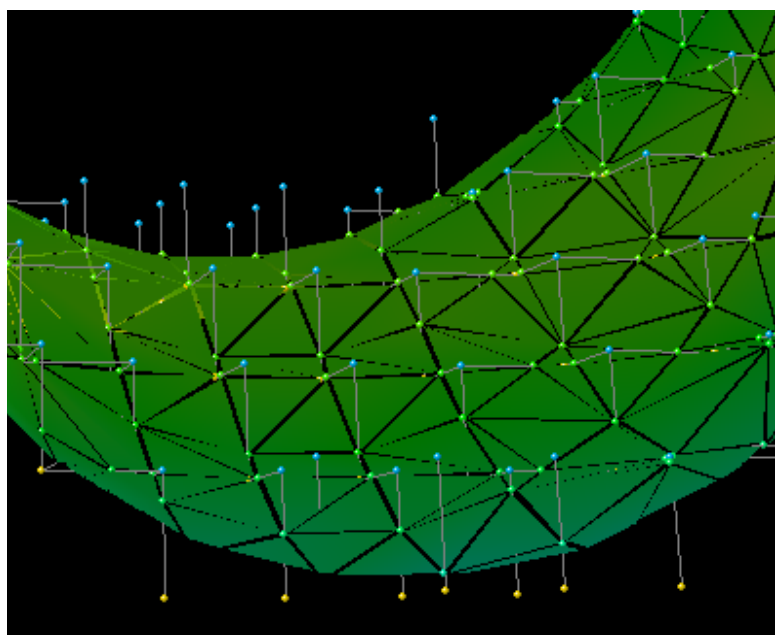


Figure 4. Triangles connect the surface-grid intersections.

The only hitch in this scheme is that the backslash character, '\', 92, is in this range. To ensure that backslash is not encoded, because it often has special meaning in strings in many programming languages, we encode backslash as an exclamation point ('!', 33). The only other character that might give problems is double quote, but it is outside this range (ASCII 34).

This, then, is all that is needed to reconstruct the critical cube data for a single isosurface, because it is this set of intersection points that form all of the vertices of the triangles used to approximate the surface (Figure 4).

The actual Java code for this value-to-character conversion is shown below:

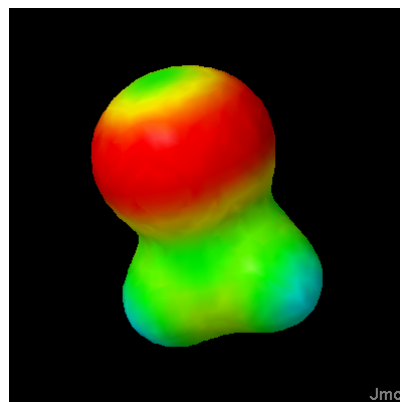
```
char jvxlFractionAsCharacter(float fraction, int base, int range) {
    if (fraction > 0.9999f)
        fraction = 0.9999f;
    else if (Float.isNaN(fraction))
        fraction = 1.0001f;
    int ich = (int) (fraction * range + base);
    if (ich < base)
        return (char) base;
    if (ich == 92)
        return 33; // \ --> !
    return (char) ich;
}
```

And its reverse, character-to-value, where fracOffset is 0.5 to center the returned fraction among the possible values assignable to that character:

```
float jvxlFractionFromCharacter(int ich, int base, int range, float fracOffset) {
    if (ich == base + range)
        return Float.NaN;
    if (ich < base)
        ich = 92; // ! --> \
    float fraction = (ich - base + fracOffset) / range;
    if (fraction < 0f)
        return 0f;
    if (fraction > 1f)
        return 0.999999f;
    return fraction;
}
```

Color Mapping Data

Color-mapped isosurfaces are constructed in exactly the same way, but in this case, each surface point – defined by the crossing of an edge by the surface – is assigned a numerical value based on some *other* criterion, typically data from another CUBE file, but also possibly charge data or other data easily determined from the atomic positions or data added by the user. Coloring is based on a particular scheme or “palette” which simply requires one more “fractional” number – the fraction of the distance from “red” to “blue”. This is encoded as before; Jmol uses a 35-color “roygbiv” rainbow for its rendering, so we encode as before, this time with a color base (CB) of 35 and a color range (CR) of 35 as well.

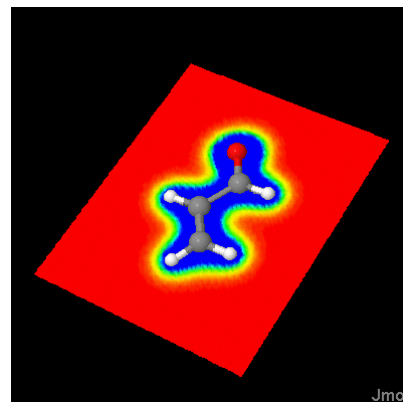


Planar Slices

Planar data are particularly amenable to compression, because we do not require any surface or edge data – the function defining the plane:

$$ax + by + cz + d = 0$$

is sufficient. In the planar version of the JVXL format we simply set both NP and NE to -1 and add the four planar parameters PA, PB, PC, and PD. But, in addition, since in this case there are typically substantial runs of the same color, we



apply a bit of simple compression. Specifically, any run of the same character “X” that is longer than 4 digits is represented by “X~n<space>”, where “n” is the number of digits removed from the run. So, for example,

```
EEEEEEB4 , ' $#####% ' , 5DEEEEEEE
```

is shortened to:

```
E~5 B4 , ' $#~33 % ' , 5DE~6
```

This allows a reduction in file size by a factor of 200-600 over delivering the raw CUBE files. (The tilde character is available for this use because it is ASCII 126, outside the 25-125 range utilized by Jmol for its character encoding.)

Solvent-Accessible and Solvent-Excluded Surfaces

Both solvent-accessible and solvent-excluded surfaces (or “molecular” surfaces) can be generated by Jmol as isosurfaces and then saved and transmitted as JVXL files. These terms have been described by Connolly. [<http://www.netsci.org/Science/Compchem/feature14e.html>] Jmol uses a method that is not described in that article. The basic idea is to create a three-dimensional grid of data points – precisely as in CUBE files – that represents the solvent surface. This is accomplished simply by defining a grid and then determining a value for each grid point for which 0 represents “on the surface”. Details of this algorithm are not described here.

Rendering of Biomolecule Cavities

The method of creating a grid of points for which 0 represents the surface to render also lends itself to the rendering of “cavities” – pockets or voids in a protein that might represent an active binding site. The algorithm Jmol uses for a general solvent surface assigns a positive value to points outside the surface. These values are the minimum distance to the surface. If a set of spheres is generated, each of which is centered on the grid points and having the specified radius, that set of “atoms” can be used in a second surface-finding pass that defines precisely the boundaries of all cavities.

Writing JVXL Files using Jmol 11.1

Creation of a JVXL file is most easily accomplished using the stand-alone Jmol Java application, version 11.1 or later. Once a dataset is visualized using the ISOSURFACE command from the console, the following command creates the JVXL file:

```
write ISOSURFACE "myfile.jvxl"
```

Alternatively, the applet can be used to create JVXL files from a browser. Clicking on the applet “Jmol” frank brings up a menu from which one can bring up a console. The command to display the contents of the JVXL equivalent of a displayed isosurface is simply:

```
show ISOSURFACE
```

The JVXL data appear in the console output window, from where they can be highlighted and clipped into the system clipboard by the user.

Results and Compression Ratios

Compression ratios on the order of 400-600 are typical for orbitals, electron density, and electrostatic potential maps. If only a planar slice is desired, compression can be much higher – on the order of 2000 – 6000 : 1. This, of course, leaves out any possibility of real-time generation of additional surfaces; for that, the CUBE file is necessary. But for general use – depicting molecular orbitals and simple mapped surfaces – JVXL files should suffice. Shown in Table 1 and Figures 5 and 6 are several comparisons.⁴

Table 1. Comparison of various surface types delivered as CUBE or JVXL files.				
Compound	Surface Type	CUBE Size/Kb	JVXL Size/Kb	Byte Ratio
CH ₃ Cl	Electron density	1813	3.5	518
CH ₃ Cl	Electrostatic Potential	1813	4.8	377
CH ₃ Cl	ESP-mapped electron density	3626	6.1	594
ethene	Molecular Orbital	1015	5.5	184
1cm	Solvent-accessible (23 x 23 x 23)	160	3.7	43
1cm	Molecular (63 x 63 x 63)	3250	32.1	100

DOCUMENT HISTORY:

- original document 5/25/06
- figure 3 corrected; caption enhanced, reference 4 added 5/25/06
- 5/26/06 – added figures and plane color compression description.
- Revised June 15, 2006
- Revised March 20, 2007
- Table added June 28, 2007

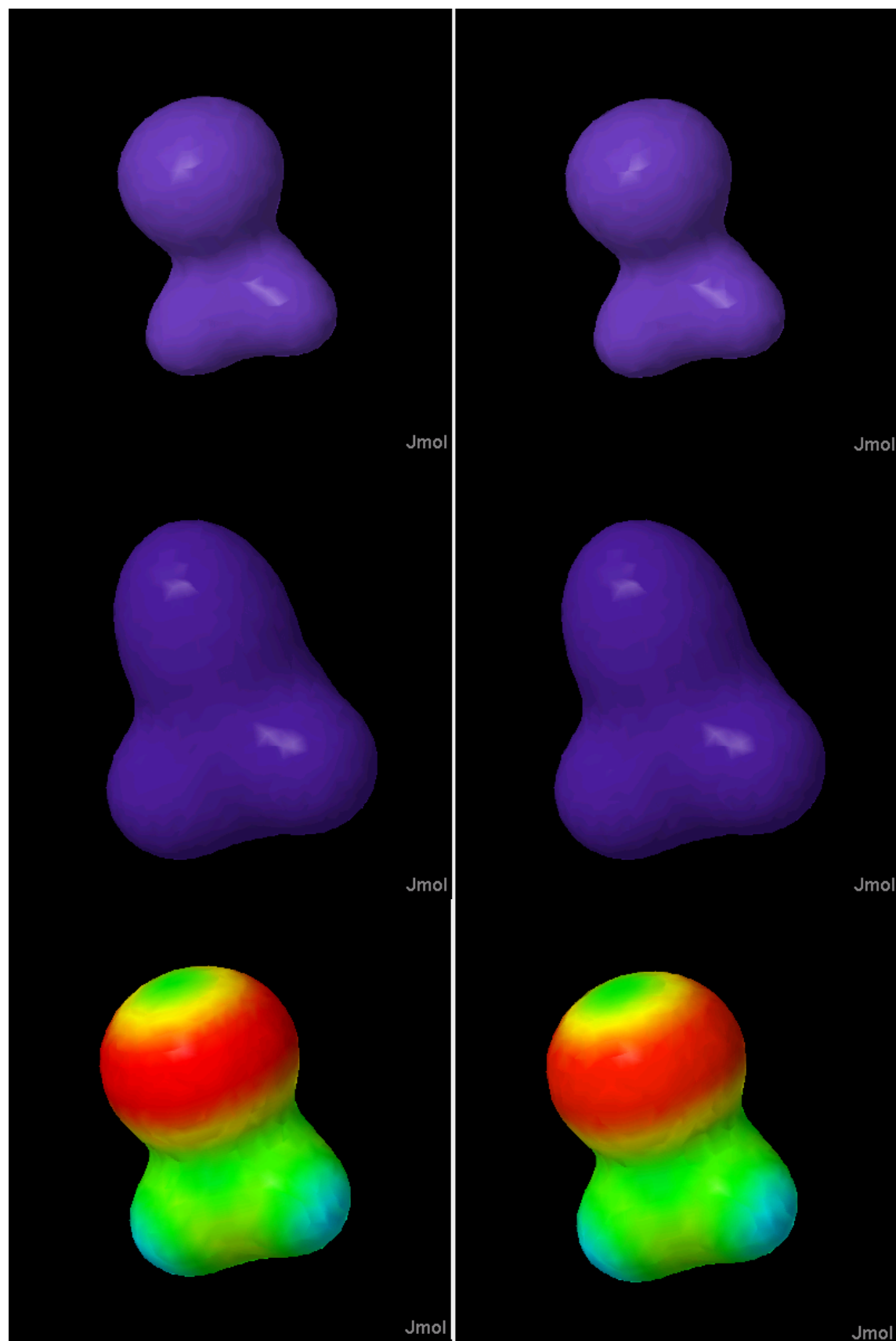


Figure 5. Top: electron density isosurface; middle: electrostatic potential isosurface; bottom: electrostatic potential mapped onto the electron density isosurface. CUBE file (left, 3.7 Mb), and JVXL file (right, 6 Kb top and middle; 7 Kb bottom).

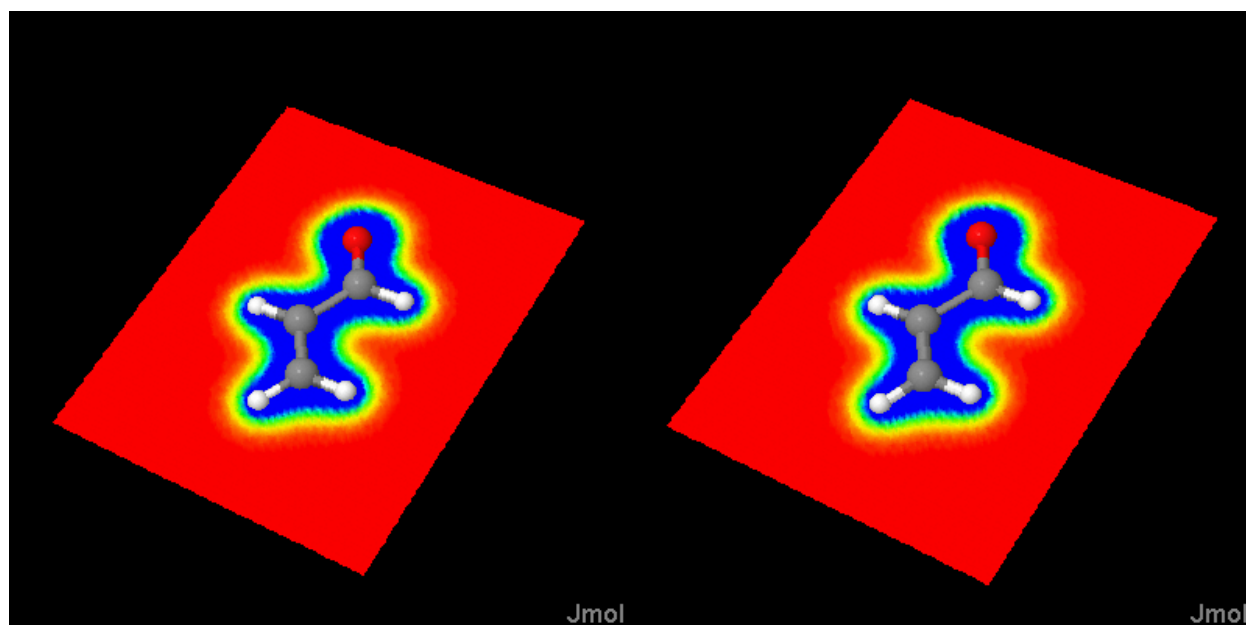


Figure 6. On the left, a 6.7 Mb acrolein CUBE file; on the right, the 2.7 Kb JVXL extraction of this particular plane of data, amounting to a 2400:1 reduction in file size. (They are for all practical purposes identical.)

References

- ¹ http://educ.gaussian.com/visual/Diff/Files/acrolein1_gs.cube (6.7Mb uncompressed).
- ² http://svn.sourceforge.net/viewcvs.cgi/*checkout*/jmol/trunk/Jmol-datafiles/cube/benzene-homo.cub.gz (568Kb gzip compressed, 1.6Mb uncompressed).
- ³ <http://www.exaflop.org/docs/marchcubes/> (last accessed 5/26/06).
- ⁴ Clockwise from the top left (Jmol script after loading given in parentheses); see <http://www.stolaf.edu/academics/chemapps/jmol/docs/examples-11/isosurface.htm>, <http://www.stolaf.edu/academics/chemapps/jmol/docs/examples-11/data/ch3cl-density.cub.gz> (isosurface 0.05 ""), <http://www.stolaf.edu/academics/chemapps/jmol/docs/examples-11/data/ch3cl.jvxl> (isosurface "" 1), <http://www.stolaf.edu/academics/chemapps/jmol/docs/examples-11/data/ch3cl.jvxl> (isosurface "" 2), <http://www.stolaf.edu/academics/chemapps/jmol/docs/examples-11/data/ch3cl-map.jvxl> (isosurface ""), <http://www.stolaf.edu/academics/chemapps/jmol/docs/examples-11/data/ch3cl-density.cub.gz> (isosurface "" color "http://www.stolaf.edu/academics/chemapps/jmol/docs/examples-11/data/ch3cl-esp.cub.gz"), <http://www.stolaf.edu/academics/chemapps/jmol/docs/examples-11/data/ch3cl-esp.cub.gz> (isosurface 0.05 "")