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

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. It 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, allowing for a far more efficient use of bandwidth. 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. [refs] 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 acrolein[1] and benzene[2]. 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.

General JVXL File Format

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). Units are Bohrs. *N must be negative; it cannot be zero.*	-5 -8.140940 -8.140940 -8.643459 -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. *There must be at least one atom.*	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 (EB, ER, CB, CR) described below, and any additional text, possibly indicating the file format version	-1 35 90 35 35 Jmol voxel format version 0.9b -NS EB ER CB CR

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" header line will be present. The negative number on that 7+N 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 JXVL 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 variable.

TABLE 2. JVXL FILE SURFACE SPECIFICATION

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 *surface voxel bitmap data* (described below)	`115922 2 6333 4 91 6 90 7 88 8 88 8 89 6 91 . . .`
One or more lines of ASCII-encoded *edge fraction data* (described below)	`3_+I6B3qPV4LVSwC{/K'_+G]fcUy6Il<3; . . .`
One or more lines of ASCII-encoded *color mapping data* (described below)	`7777889989:;;::<==:<=###$####$$%22223333###$. . .`

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 rectangular or parallelepiped grid. An isosurface is defined as a surface through these points where a given parameter has a constant value (Figure 1).

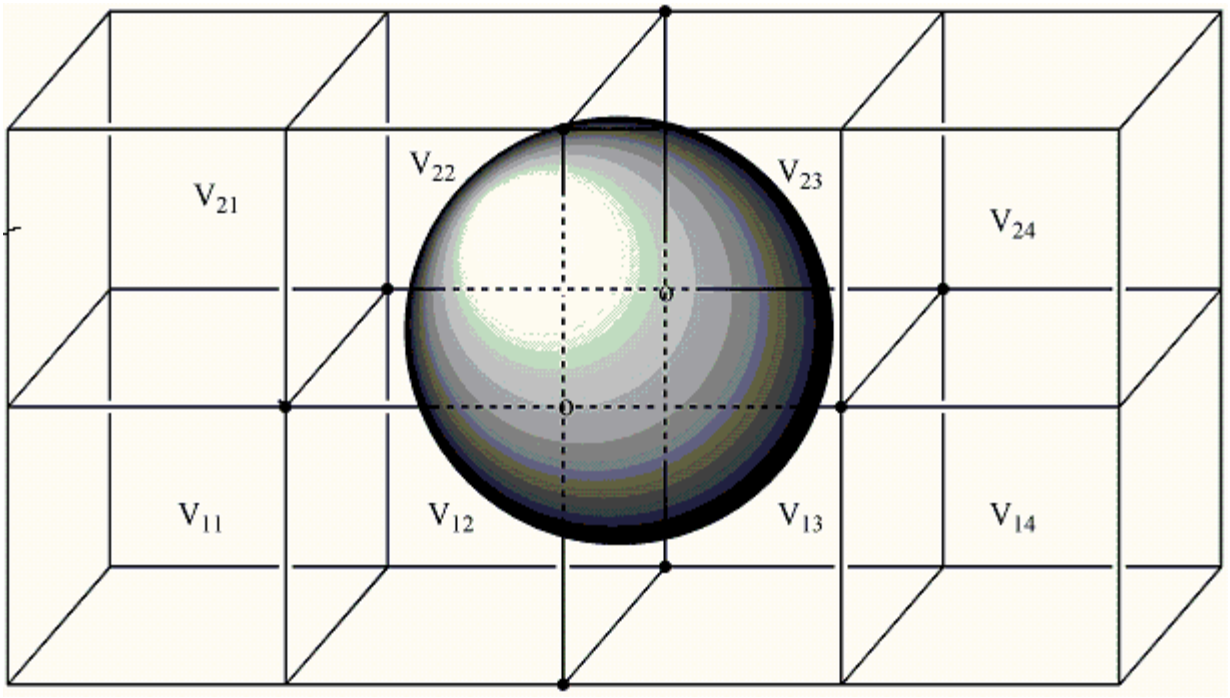

Figure 1. An isosurface surrounded by a 5 x 2 x 3 grid of data points. The key point is that much of the data points are unnecessary.

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.

The first set of surface data maps out which data points are inside and which 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 2) using a clever method that never checks any edge twice.

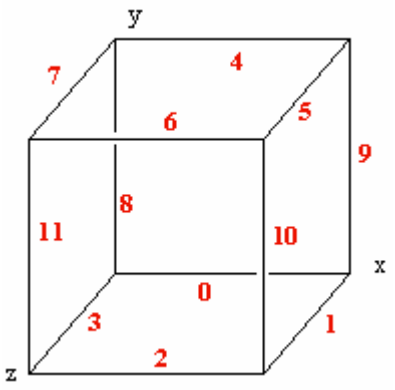

Figure 2. Edge numbering.

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:

```
fraction = (cutoff - valueA) / (valueB - 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). 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.

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. Coloring is based on a particular scheme or "pallete" 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.

Solvent-Accessible Surfaces

It should also be possible to quite simply construct isosurfaces related to solvent accessibility. In the case of solvent-accessible surfaces, the surface would be generated by a network of points that lie within the voxels. These points would be identified by other algorithms, each representing the solvent sphere in a different location. Once these points are determined, the vertex values would simply be assigned a value based on their distance from that particular solvent center. Voxels further than the solvent radius from the solvent center would be "inside" the surface; voxels nearer to the solvent center than its radius would be "outside". Provided that point density is high enough, a continuous set of voxels will be generated, and any missing voxels could, in principle, be interpolated from these.

Results and Compression Ratios

Compression ratios on the order of 400-600 are typical. 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 Figure 3 are several comparisons. [4]

DOCUMENT HISTORY:

- original document 5/25/06
- figure 3 corrected; caption enhanced, reference 4 added 5/25/06

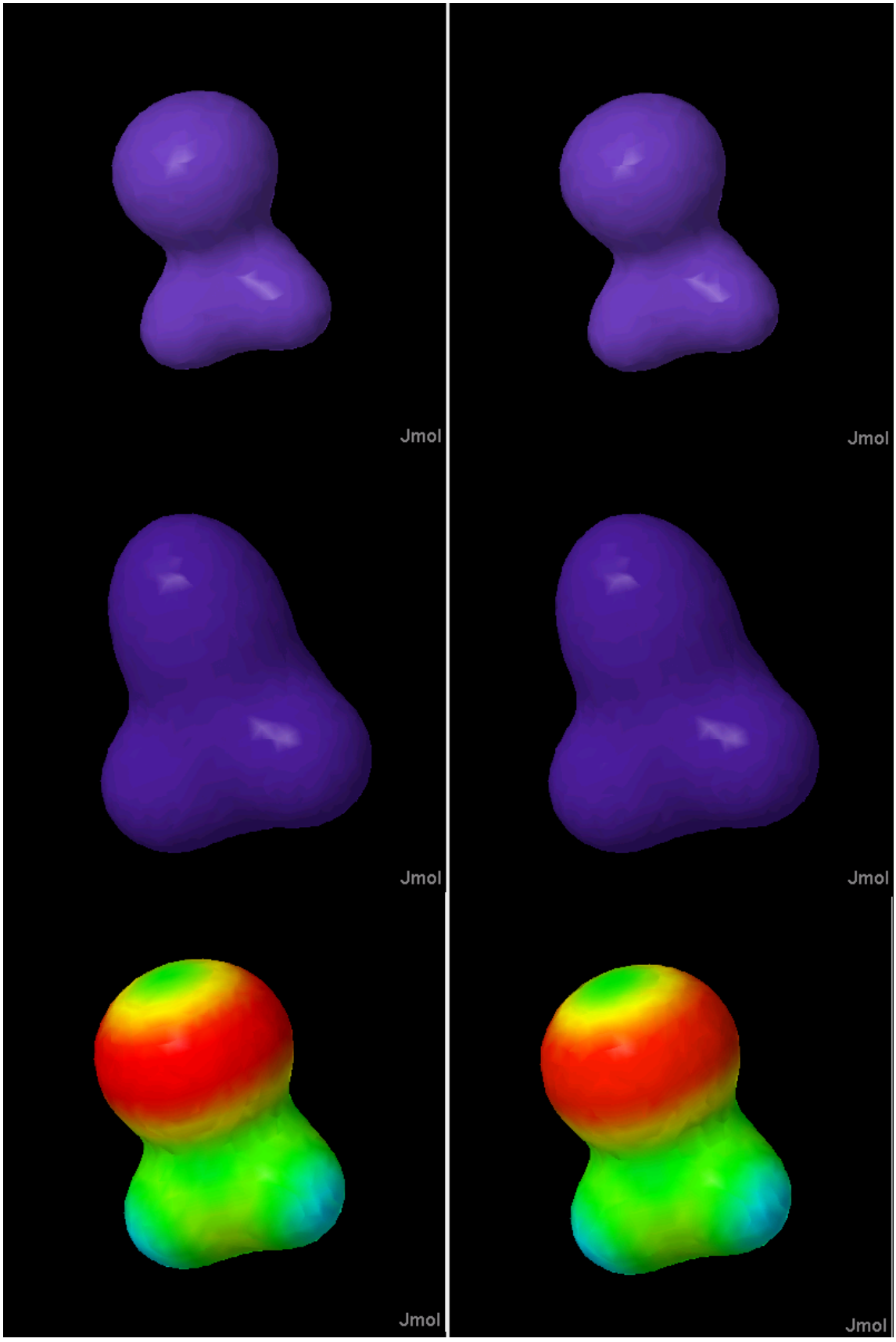

Figure 3. Top: electron density isosurface; middle: electrostatic potential isosurface; bottom: electrostatic potential mapped onto the electron density isosurface. CUBE file (left, 3.7 Mb), and JVXL file (right, 6 Kb top and middle; 7 Kb bottom).

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/people/hansonr/jmol/test/proto/new.htm:
http://www.stolaf.edu/people/hansonr/jmol/test/proto/ch3cl-density.cub.gz (isosurface 0.05 “”),
http://www.stolaf.edu/people/hansonr/jmol/test/proto/ch3cl.jvxl (isosurface fileindex 1 “”),
http://www.stolaf.edu/people/hansonr/jmol/test/proto/ch3cl.jvxl (isosurface fileindex 2 “”),
http://www.stolaf.edu/people/hansonr/jmol/test/proto/ch3cl-map.jvxl (isosurface “”),
http://www.stolaf.edu/people/hansonr/jmol/test/proto/ch3cl-density.cub.gz
(isosurface “” color “http://www.stolaf.edu/people/hansonr/jmol/test/proto/ch3cl-esp.cub.gz”)
http://www.stolaf.edu/people/hansonr/jmol/test/proto/ch3cl-esp.cub.gz (isosurface 0.05 “”)