

---

# *An Overview of VAPOR Data Collections*

July, 2006

Version 1.0.3

## Vapor Data Collection

This document provides an introduction to VAPOR's data storage model, the VAPOR Data Collection, and presents information needed for preparing data for visualization and analysis in the VAPOR environment.

### Overview

The VAPOR data analysis environment targets data sets that are time-varying, multivariate, and possessing very high spatial resolutions. Aggregate data sets generated from a single experiment that are terabytes in size are not uncommon. To accommodate the unique needs of these large data sets, VAPoR defines its own mechanism for storing field data and its associated attributes (metadata). In the VAPoR environment, a collection of related data, typically having been produced from a single numerical simulation, is known as a *VAPoR Data Collection* (VDC).

A VDC is composed of two components: *metadata* and *field data*. Metadata are data that describe field data. Examples of metadata include the grid type, spatial resolution, name of the field variables, number of time steps, and possibly user-defined attributes. Field data are the numerical outputs produced by the simulation. Examples include: components of a velocity field, a temperature field, etc.

**The VDC model is different from more traditional scientific data representations, such as netCDF and hdf, in two important ways:**

1. Field data are stored as wavelet transformed coefficients. I.e. field data undergo a user-defined number (and type) of wavelet transforms before they are written to a file. Inverse wavelet transforms may be easily and efficiently applied to the stored wavelet coefficients, and the original field data reconstructed. Furthermore, the data need not be reconstructed at its original grid resolution. The user may elect to reconstruct a coarsened approximation of the data to reduce memory requirements, processing time, etc.
2. VDC data (metadata and field data) are not stored in a single file as is commonly done with other scientific data formats. Instead, metadata, individual field data time steps, variables, and wavelet coefficients are all stored in separate files. Distributing the pieces in this manner is essential for effectively managing terabyte sized data collections.

## Reading and Writing Vapor Data Collections

VAPOR provides a collection of tools for importing simulation field data into a VDC, or exporting data from a VDC to external applications. Presently, importing and exporting operations may be performed via command line utilities, extensions to RSI's IDL, or through a C++ library. This document gives an overview of the available command line utilities and IDL language extensions.

Before describing the import/export operators, a basic understanding of the VDC file structure is required. As discussed above, the components of a VDC are distributed across different disk files. All metadata for a VDC are stored in a single *.vdf* file. A *.vdf* file is encoded in XML and may easily be browsed, or even edited, with widely found XML editors and browsers (most web browsers will read and display XML).

Field data are stored in netCDF data files as coefficients output from a user-directed number of wavelet transformation passes. Each *.netCDF* file contains the wavelet coefficients associated with a single wavelet transformation pass applied to a single field variable, at a single time step. For example, applying two transformation passes to the X component of a velocity vector,  $v_x$ , from the first time step in a data collection would result in the generation of three netCDF files with the extensions **.nc0**, **.nc1**, and **.nc2**. The first file, **.nc0**, contains the wavelet coefficients necessary to reconstruct  $v_x$  at its coarsest approximation level (1/4 the original resolution along each coordinate axes). The **.nc1** file provides the wavelet coefficients necessary to reconstruct the  $v_x$  variable at 1/2 the original resolution, etc. By splitting the wavelet coefficients into separate files, the user is afforded the opportunity to maintain an incomplete, but still valid VDC.

Thus a VDC is considered valid even if finer approximation levels are missing. In the above example, the user may choose to store the **.nc2** coefficients off line in order to save space. Furthermore, a VDC is valid even if entire time steps or variables are not present on disk. The goal of supporting incomplete VDCs is to provide the user the flexibility needed to manage very large data sets. Hence a minimal valid VDC consists only of a metadata *.vdf* file, and no field data.

The process of creating a VDC is straightforward:

1. Generate a *.vdf* file defining the number and name of variables, number of time steps, and resolution of each volume in a data set, as well as the number of wavelet transformations to apply.
2. Translate raw data volumes into wavelet-transformed coefficients.

The first step is performed once for a VDC. The number of variables, time steps and the resolution are all determined by the data itself. The number of wavelet transforms is a user option that determines how many, and what resolution, field data approximations will be available for subsequently transformed data. Specifying a value of zero implies no transformations and the data will only be available at full resolution. A value of one implies a single transformation; the data will be available at full and half resolution. And

so on. Step two may be repeatedly performed as needed, and when needed, until the VDC is fully populated as defined by the associated *.vdf* file.

The sections that follow describe the various tools available for importing and exporting raw data to/from a VDC.

### **Command line utilities**

Three command line utilities are currently available for writing and reading a VDC:

<b>Command</b>	<b>Description</b>
<b>Vdfcreate</b>	Generates a <i>.vdf</i> metadata file
<b>Raw2vdf</b>	Forward transforms a file containing a block of floats and stores the results in a VDC
<b>Vdf2raw</b>	Inverse transforms a field variable found in a VDC and stores the results on disk as a block of floats.

A brief overview of each command is presented below. For a complete description of all of the command line options supported by a command, the command may be invoked with the ‘-help’ option. For example:

```
vdfcreate -help
```

Note: Prior to running any VAPOR commands you must configure your user environment by sourcing one of the VAPOR setup scripts: **vapor-setup.csh** or **vapor-setup.sh**. For C-shell or its derivatives use:

```
source vapor-setup.csh
```

For the Bourne shell or its derivatives use:

```
. vapor-setup.sh
```

### **vdfcreate**

The **vdfcreate** command line utility generates a *.vdf* file defining a VDC. The user may specify a limited number of metadata attributes including: the volume dimension, number of time steps, number of forward wavelet transforms, and the variable names. The invocation:

```
vdfcreate -dimension 512x512x512 -numts 100 \  
-level 3 -varnames vx:vy:vz foo.vdf
```

would produce a *.vdf* file named **foo.vdf**. This file would describe a VDC containing 100 time steps, starting from 0 and running through 99; three field variables, **vx**, **vy** and **vz**;

each volume would have a spatial resolution of  $512^3$ ; and three wavelet transforms would be applied. Thus the data will be accessible at the following resolutions:  $512^3$ ,  $256^3$ ,  $128^3$ , and  $64^3$ .

## raw2vdf

The **raw2vdf** command reads a volume from disk stored as a block of floats (a contiguous, 3D array of unformatted, 32bit, binary floating point values with no header or trailer information), transforms the data into wavelet space, and stores it in a VDC. Assuming we used the *.vdf* file created in the previous example, the command:

```
raw2vdf -ts 0 -varname vx foo.vdf rawvx.float
```

would transform the volume stored in the file **rawvx.float** and write it into the VDC associated with the **foo.vdf** metafile. The time step and variable would be 0 and vx, respectively. The volume contained in **rawvx.float** must have a resolution of  $512^3$  as defined in the *.vdf* file. Furthermore, the volume will undergo three wavelet transformations, resulting in a coarsest resolution of  $64^3$ .

Note: See the Appendix for a discussion on converting data output from a FORTRAN program into a VDC.

## vdf2raw

This command extracts a variable (3D data volume) from a VDC, applies an inverse wavelet transform, and stores the results in a file. Following our previous example, the command:

```
vdf2raw -ts 0 -varname vx -level 2 foo.vdf \  
rawvx256.float
```

would extract a  $256^3$  version of the vx variable, at time step 0, and store the inverse wavelet transformed results as a block of floats in the file rawvx256.float.

## IDL Language Extensions

Extensions to the IDL language in the form of new IDL system routines provide a powerful mechanism for interacting with VDCs, offering data interaction capabilities that go far beyond what is currently possible with the suite of command line utilities described above. Using IDL language extensions a user may not only define a *.vdf* file, as with the command line utilities, but may also programmatically edit an existing file, or read the contents of a *.vdf* file into an IDL session. Furthermore, the IDL language extensions expose *.vdf* metadata attributes that are not accessible via the command line utilities. For example, with the IDL routines a user may set or get user-defined floating, integer, or character text arrays associated with a time step, a variable within a time step, or that are global to the entire VDC. These user-defined attributes have no meaning to VAPOR. Their interpretation and application is entirely up to the user.

In addition to operating on *.vdf* files, the IDL routines also provide a number of mechanisms for importing/exporting data to/from a VDC. Field data imported from a VDC into IDL are stored as a 3D IDL array. Similarly, IDL export routines export data contained in a 3D IDL array.

Note: Prior to running any VAPOR IDL commands you must configure your user environment by sourcing one of the VAPOR setup scripts: **vapor-setup.csh** or **vapor-setup.sh**. For C-shell or its derivatives use:

```
source vapor-setup.csh
```

For the Bourne shell or its derivatives use:

```
. vapor-setup.sh
```

### A simple IDL session

The following step-by-step instructions demonstrate a simple IDL session using one of the IDL example scripts provided by VAPOR. It assumes that the user's VAPOR configuration environment has been properly established as per above.

1. Change working directories to the VAPOR IDL examples directory:

```
% cd $VAPOR_HOME/examples/idl
```

2. Invoke the IDL interpreter using whatever IDL startup command is appropriate for your environment. For example, simply typing `idl` will work in many environments:

```
% idl
```

3. From the IDL command prompt, run the example program named **WriteVDF.pro**:

```
IDL> .run WriteVDF
```

4. Exit the IDL interpreter:

```
IDL> exit
```

You should now have a VAPOR data collection in the /tmp directory named test.vdf. This data set may be explored using VAPOR's interactive data browser, **vaporgui**.

## IDL Example Scripts

A number of rudimentary IDL example scripts are presented below.

### Creating a .vdf file – example 1

IDL system routines for operating on .vdf files are all prefixed with a *vdf\_*, for example, *vdf\_create*. The following example shows how to create a simple .vdf file that will contain only a single time step, and a single variable.

```
;
; Dimensions of the data volumes - all volumes
; in a dataset must be of the same dimension
;
dim = [512,512,512]

;
; Number of forward wavelet transforms to apply to
; data stored in the data set. A value of 0 indicates
; that the data should not be transformed. A value of 1
; implies a single transform. The resulting
; data will be accessible at the original resolution
; and 1/8th resolution(half resolution in each dimension).
; A value of two implies two coarsenings, and so on.
;
num_levels = 2

;
; Create a new VDF metadata object of the
; indicated dimension and transform level. vdf_create()
; returns a handle, named mfd in this example,
; for future operations on the metadata object.
;
mfd = vdf_create(dim,num_levels)

;
; Set the maximum number of timesteps in the data set.
; Note, a valid data set may contain less than the
; maximum number of time steps, but not more
;
timesteps = 1
vdf_setnumtimesteps, mfd,timesteps

;
; Set the names of the variables the data set will
; contain. In this case, only a single
; will be present, "vx".
```

```
;
varnames = [vx]
vdf_setvarnames, mfd, varnames

;
;   Store the metadata object in a file for subsequent use
;
vdf_file = '/tmp/test.vdf'
vdf_write, mfd, vdf_file

;
; Destroy the metadata object. We're done with it.
; Note. Neglecting to destroy the object once you
; are done with it can have unpredictable results.
vdf_destroy, mfd
end
```

## Creating a .vdf file – example 2

This example differs slightly from the first. We'll add more time steps, variables, and set some other metadata attributes.

```
dim = [512,512,512]
num_levels = 2

mfd = vdf_create(dim,num_levels)

;
; This VDC will contain 100 time steps
;
timesteps = 100
vdf_setnumtimesteps, mfd,timesteps

;
; Set the names of the variables the data set will
; contain. In this case, three variables
; will be present, "vx", "vy", and "vz"
;
varnames = ['vx',vy,vz]
vdf_setvarnames, mfd, varnames

;
; Set the extents of the volume in user-defined
; physical coordinates. Note, as the aspect ratio of
; the user-defined coordinates do not match that of
; the volume resolution (512x512x512), the volume
; will be stretched when rendered. I.e. the spacing
; between the Z coordinate samples is defined to
; be twice that of X or Y.
;
extents = [0.0,0.0,0.0,100.0, 100.0, 200.0]
vdf_setextents, mfd, extents

;
; Set a global comment
;
vdf_setcomment, mfd, 'This is my xxx data'

;
; Set user defined, floating point attribute
; data. In this case, monotonically increasing floating
; point numbers from 0 to 99.0.
;
; We first define a name for the attribute, "MyMetadata".
```

```
;
attribute_name = 'MyMetadata'
f = findgen(100)
Vdf_setdbl,mfd,attribute_name, f

vdf_file = '/tmp/test.vdf'
vdf_write, mfd, vdf_file

vdf_destroy, mfd
end
```

## Writing field data

Once a *.vdf* file has been created using either the IDL commands discussed above, or the command line utilities discussed previously, the VDC may be populated with data. IDL routines that operate on field data are prefixed with *vdc\_*.

Note: See the Appendix for a discussion on converting data output from a FORTRAN program into a VDC.

```
;
; Create a "buffered write" data transformation
; object. The data transformation object will permit
; us to write (transform) raw data into the data set.
; The metadata for the data volumes is obtained
; from the metadata file we created previously. I.e.
; 'vdffile' must contain the path to a previously
; created .vdf file. The vdc_bufwritecreate routine
; returns a handle, 'dfd', for subsequent operations.
;
dfd = vdc_bufwritecreate(vdffile)

; Get the data volume that we wish to store.
;
vx = my_data_generation_function()

;
; Prepare the data set for writing. We need to identify
; the time step and the name of the variable that
; we wish to store. In this case, the first time step,
; zero, and the variable named 'vx'
;
vdc_openvarwrite, dfd, 0, 'vx'

;
; Write (transform) the volume to the data set one
; slice at a time
;
for z = 0, dim[2]-1 do begin
    vdc_bufwriteslice, dfd, vx[*,*,z]
endfor

;
```

```
; Close the currently opened variable/time-step. We're
; done writing to it
;
vdc_closevar, dfd

;
; Destroy the "buffered write" data transformation
; object. We're done with it.
;
vdc_bufwritedestroy, dfd

end
```

## Reading data – example 1

Reading data from a VDC is similar to writing:

```
; This example shows how to read a single data volume
; from a VDC data collection using a "Buffered Read"
; object. The entire spatial domain of the volume
; is retrieved (as opposed to fetching a spatial
; subregion). However, the volume is extracted at
; its coarsest spatial resolution.
;
;
; Number of forward wavelet transforms.
; A value of 0 indicates that the data should be
; read at coarsest) resolution. A value of 1
; implies a the next coarsest, etc. A value of -1
; implies the finest (native) data resolution.
;
num_levels = 0

vdfilename = '/tmp/test.vdf'

;
; Create a "Buffered Read" object to read the data,
; passing the path to the metadata file created in;
; the previous example.
;
dfd = vdc_bufreadcreate(vdfilename)

;
; Determine the dimensions of the volume at
; the given transformation level.
;
; Note. vdc_getdim() correctly handles dimension
; calculation for volumes with non-power-of-two dimensions.
;
dim = vdc_getdim(dfd, num_xforms)

;
; Create an appropriately sized array to hold the volume
; and a 2D array for reading the data
;
f = fltarr(dim)
slice = fltarr(dim[0], dim[1])
```

```

;
; Prepare to read the indicated time step and variable
;
vdc_openvarread, dfd, 0, 'vx', num_levels

;
; Read the volume one slice at a time
;
for z = 0, dim[2]-1 do begin
    vdc_bufreadslice, dfd, slice

        ; IDL won't let us read directly into a
        ; subscripted array - need
        ; to read into a 2D array and then copy to 3D ☹
        ;
        f[*,*,z] = slice
endfor

;
; Close the currently opened variable/time-step.
;
vdc_closevar, dfd

;
; Destroy the "buffered read" data transformation object.
; We're done with it.
;
vdc_bufreaddestroy, dfd

end

```

## Reading data – example 2

This example differs from the previous one in that we extract a spatial subregion of the volume instead of reading the entire volume.

```
; Set the refinement level. A value of -1 implies
; native data resolution.
;
num_levels = -1

;
; Create a "Region Read" object to read the data.
; The handle returned by this routine will permit
; data to be read from the data collection associated
; with the .vdf file, '/tmp/test.vdf'
;
vdf_file = '/tmp/test.vdf'
dfd = vdc_regreadcreate(vdf_file)

dim = vdc_getdim(dfd, num_levels)

;
; Compute the coordinates for the desired subregion.
; In this case, the first octant will be read
min = [0,0,0]
max = (dim / 2) - 1

;
; Create an appropriately sized array to hold the volume
;
vx = fltarr(dim/2)

vdc_openvarread, dfd, 0, 'vx', num_levels

;
; Read the volume subregion. Note, unlike the
; buffered read/write objects, the "Region Reader"
; object does not read a single slice
; at a time -- it slurps in the entire region
; in single call.
;
vdc_regread, dfd, min, max, vx

vdc_closevar, dfd

vdc_regreaddestroy, dfd
end
```

## Appendix A – Handling data generated by a Fortran program

Getting data from a Fortran code into a VDC can be tricky business. At present an API for writing output directly from a Fortran program to a VDC is not supported. Hence, data must first be written out to an intermediary file, and subsequently translated to a VDC using either command line tools or the IDL interface discussed earlier. The tricky part is in generating an intermediary file that the conversion utilities can understand. Two issues that arise are:

- Differences in bit *endianness* between the machine generating the data and the machine where the data translation will take place
- Arcane syntax required by Fortran to output a raw data file.

### **Bit Endianness**

Two machine representations for binary floating point numbers are commonly found today: Little-endian (used by Intel processors) and Big-endian (used by most everybody else). If the raw data file generated by your simulation code was produced on a machine different from the one you are creating your VDC on, there may be an endian mismatch. Fortunately, this is easily resolved by using either the `-swapbyte` switch, if using the `raw2vdf` command line utility, or using the `/SWAP_ENDIAN`, `/SWAP_IF_LITTLE_ENDIAN`, or `SWAP_IF_BIG_ENDIAN` `open` procedure keywords, if using the IDL language extensions, provided by VAPOR, to translate your data.

### **Writing raw data from Fortran**

Creating a raw data file from Fortran - one containing binary data with no header or trailer – requires more effort than it should. Common practice among many Fortran programmers is to write binary data as an unformatted, sequential file. Unfortunately, on UNIX systems this results in the inclusion of a data header and/or trailer. Care must be taken if the header is to be avoided. The code snippet below demonstrates how to write a contiguous volume of data (3d array) as a raw file:

```
REAL MYARRAY (NX, NY, NZ)

OPEN(1, FILE= 'myarray.raw' , FORM= 'UNFORMATTED' ,
ACCESS= 'DIRECT' , RECL=NX*NY*NZ)
WRITE (1, REC=1) MYARRAY
CLOSE (1)
```

The resulting file, `myarray.raw`, may now be translated using either IDL or the command line utilities described earlier.