Using Python with VAPOR
March 2015 update (VAPOR 2.4)

Introduction:

VAPOR Version 2.4 includes support for using Python (version 2.7.8), with the Numerical Python module (NumPy version 1.8.1) and SciPy version 0.14.0, to define derived variables in VAPOR. New variables can be defined as the result of applying a python script to other variables, and these new variables are automatically calculated as needed during the visualization session.

Basic usage:

New variables can be defined as the result of an array-based operation. A mathematical formula can be applied to existing variables to define new variables. For example, suppose U and V are 3D variables in a Vapor Data Collection (VDC). U and V are represented in VAPOR as collections of 3D data arrays, one array for each time step. You can define a new 3D variable "NewVar" with the one-line Python script:

NewVar = 2.5*U +10***U***V

Thereafter the variable "NewVar" can be used as a 3D variable in any 3D VAPOR visualization. Its value at a point (x,y,z) in the VAPOR scene, and at a time step t, will be determined by applying the above formula to the corresponding point of the U and V arrays for the same time step t. You can also calculate a variable by operating on input data arrays element-by-element; this is discussed below under "Advanced Usage".

Vapor's data caching and region selection mechanisms are applicable, so that NewVar will only be re-calculated as needed.

To define a new variable that is derived from other variables, do the following:

1. From the vaporgui Edit menu, select "Edit Python program defining a new variable". This will launch the VAPOR Python editor.

2. From the VAPOR Python editor, check the existing variables that will be used as input to the script, using the checkboxes at the top of the editor window. In the above example, you would check "U" and "V" as Input 3D Variables.

3. Specify the name(s) of output variables calculated by your script, using the "Add 3D Variable" or "Add 2D Variable" pushbuttons in the editor window. In the above example, "NewVar" would be specified as an Output 3D variable.

4. Type the formula (or Python program) that calculates your output variable(s) into the center Python editor window. You can also copy and paste it from another text editor.

5. Optionally, click on the "Test" button at the bottom of the editor window. This will check that your program is valid by applying it at the current region and time step, at lowest refinement level, and will print any output text or error messages in a VAPOR popup.

6. Click on the "Apply" button to finalize the program, defining the new variable. Henceforth the newly defined variable(s) will appear in the various variable lists for rendering.

It is a good idea to save your session after you click "Apply". That way the derived variables and the Python scripts you have defined will be available the next time you load the session.

Pre-defined Python functions

VAPOR provides several Python functions that are available from the VAPOR environment. General functions are in the module "vapor_utils". WRF-specific modules are in the module "vapor_wrf". The functions in vapor_utils are by default imported. You can access the functions in vapor_wrf by issuing "import vapor_wrf" in your python shell.
Help is available in the vapor python editor. To obtain help on method named "`method_name`" in module "`module_name`", type the following into the python editor:

```
import module_name
help (module_name.method_name)
```

Then click the "Test" button.
On Macintosh the python methods are by default installed in Vapor.app/Contents/SharedSupport/python. On Windows or Linux these are installed in $(VAPOR_HOME)/share/python . You may also find these scripts useful examples of how to write Python scripts operating on VAPOR data.

The **vapor_utils** module includes:

vapor_utils.mag3d(a1,a2,a3) : return the 3D magnitude of (a1,a2,a3)

vapor_utils.mag2d(a1,a2) : return the 2D magnitude of (a1,a2)

vapor_utils.curl_findiff(inx, iny, inz, order=6): returns (a tuple of) 3 NumPy arrays corresponding with the three components of the curl of the vector field (inx,iny,inz). The curl is performed in VAPOR user coordinates, so that the results may be used in the current VAPOR visualization. Order can be 6, 4, or 2.

vapor_utils.grad_findiff(var,order=6): returns (a tuple of) 3 NumPy arrays corresponding with the three components of the gradient of the variable var. The gradient is performed in VAPOR user coordinates, so that the results may be used in the current VAPOR visualization. Order can be 6, 4, or 2.

vapor_utils.div_findiff(inx,iny,inz,order=6): returns a NumPy array which is the divergence of the vector field defined by [inx, iny, inz]. The divergence is performed in VAPOR user coordinates, so that the results may be used in the current VAPOR visualization. The order can be 2,4,or 6(default).

vapor_utils.deriv_findiff(a,dir,dx,order=6): return the finite-difference (sixth order) derivative of the NumPy float32 array a, in the direction dir, with a spatial step of dx. The value of dir is 1,2, or 3, which are associated with the 1st, 2nd, and 3rd dimensions of the NumPy array, and which correspond to the Z, Y, and X dimensions in the VAPOR VDC. The floating point value dx is the step-size in user coordinates associated with grid size in the dimension associated with dir. The order can be either 6, 4, or 2.

vapor_utils.deriv_var_findiff(a,var,dir,order=6): return the finite-difference (sixth order) derivative of the NumPy float32 array a, with respect to the variable "var". The value of dir is 1,2, or 3, which are associated with the 1st, 2nd, and 3rd dimensions of the NumPy array, and which correspond to the Z, Y, and X dimensions in the VAPOR VDC. The order can be either 6, 4, or 2.

vapor_utils.interp3d(A,PR,val): returns a 2D horizontal variable, obtained by interpolating the 3D variable A to the level determined by PR=val, where PR (typically pressure) is a 3D variable that decreases with increased elevation (z) and val is a scalar.

vapor_utils.vector_rotate(angleRad, latDeg, u, v): returns a 2-tuple of variables obtained by rotating the u and v variables to a vector field on a latitude/longitude grid. The variable angleRad specifies the angle rotation in radians to convert from the u,v grid to the lat/lon grid. The latDeg variable indicates the latitude in degrees. This is useful for performing flow integration on the vapor lat-lon grid, with ROMS, MOM, POP, and other models that are reinterpolated to a lat-lon grid for visualization. The horizontal components of the flow must be rotated and scaled for correct flow integration on the lat-lon grid.

VAPOR user preference settings include a "Python directory", where users can save the python scripts they are using. By default, on Linux the Python directory is the current directory. On Mac and Windows it defaults to the user's home directory (the value of the HOME environment variable).

Users may also provide a Python startup script for defining methods, variables and other settings to be used in a particular session. From the Edit menu, select "Edit

Python startup script". Click "Test" to ensure the script works in the VAPOR environment. After you click "Apply", this script will be executed once, prior to any other scripts. The startup script will be saved in the session file when you save the session, and will be restored when you restore a session.

When VAPOR is installed, a Python system startup script, named "pythonSystemStartup.py", is installed in the directory $(VAPOR_HOME)/share/python . The system startup script is executed prior to any other python commands, and is present in all VAPOR sessions. By default, this script contains only the two lines:

```
from numpy import *
from vapor_utils import *
```

The vapor_wrf module

Note: This module was extensively updated for vapor 2.2. Previous to VAPOR 2.2, python methods operated in the user coordinate grid, which was reinterpolated from the WRF terrain-following grid. With VAPOR 2.2, the python methods operated directly on arrays defined on the WRF grid. This results in more accurate calculation, however modifications whenever using derivative operators to take into account the grid layering. New methods are provided in this module to support derivative operators.

Several methods that are commonly used in analysis of WRF-ARW output are installed with vapor, in the vapor_wrf module. To use these you should include the statement "import vapor_wrf" at the start of your script. These operate on the standard WRF output variables. They have been adapted from RIP4 and NCL FORTRAN-based utilities. Included are:

vapor_wrf.CTT(P,PB,T,QCLOUD,QICE) : (2D) cloud-top temperature

vapor_wrf.CTHGT(P,PB,T,QCLOUD,QICE) : (2D) cloud-top height.

vapor_wrf.DBZ(P,PB,QRAIN,QGRAUP,QSNOW,T,QVAPOR,iliqskin,ivarint): Calculates 3D radar reflectivity. Optional variables iliqskin and ivarint default to 0. If iliqskin = 1, then frozen particles above freezing are assumed to scatter as a liquid particle. If ivarint = 1 then intercept parameters are calculated based on the work of Thompson, Rasmussen and Manning in 2004 Monthly Weather Review.

vapor_wrf.DBZ_MAX(P,PB,QRAIN,QGRAUP,QSNOW,T,QVAPOR, iliqskin,ivarint): Calculates 2D Radar reflectivity; i.e. the vertical maximum of DBZ, as described above.

vapor_wrf.ETH(P,PB,T,QVAPOR): Calculates the 3D equivalent potential temperature.

vapor_wrf.PV(T,P,PB,U,V,ELEVATION,F): Calculates the 3D potential vorticity. Note that F is a 2D WRF variable. ELEVATION is the VAPOR 3D variable that specifies the elevation in meters above sea level.

vapor_wrf.RH(P,PB,T,QVAPOR): Calculates the 3D relative humidity.

vapor_wrf.SHEAR(U,V,P,PB,level1=200,level2=850): Calculates horizontal wind shear, defined as the RMS difference between the horizontal velocity, interpolated to the specified pressure levels in millibars(level1 and level2)

vapor_wrf.SLP(P,PB,T,QVAPOR,ELEVATION): Calculates the 2D sea-level pressure. Note that ELEVATION is the VAPOR-WRF 3D variable indicating vertical elevation above sea level

vapor_wrf.TD(P,PB,QVAPOR): Calculates 3D dewpoint temperature

vapor_wrf.TK(P,PB,T): Calculates 3D temperature in degrees Kelvin.

Note: The following four methods are useful whenever dealing with layered grids, such as WRF or ROMS.

vapor_wrf.wrf_deriv_findiff(A,ELEV,dir): Calculates the derivative of a WRF (or layered) variable in the direction dir (=0,1, or 2), where ELEV is the VAPOR ELEVATION variable.

vapor_wrf.wrf_grad_findiff(A,ELEV): Returns the gradient of a scalar field A, where ELEV is the VAPOR ELEVATION variable.

vapor_wrf.wrf_div_findiff(A,B,C,ELEV): Returns the divergence of the vector field (A,B,C) where ELEV is the VAPOR ELEVATION variable. The result is a 3-dimensional variable representing the divergence of (A,B,C).

vapor_wrf.wrf_curl_findiff(A,B,C,ELEV): Returns the curl of the vector field (A, B, C) where ELEV is the VAPOR ELEVATION variable. The result is a 3-tuple of 3-dimensional arrays.

Python output text:

The Python editor in VAPOR does not have all the normal functionality of a Python interpreter. Users will not see any results or messages until they press “Test” or “Apply”. If the script has no output text and no errors, there is a popup indicating successful execution. To debug these Python scripts it is often useful to print some results as well as other diagnostic information. To provide such feedback, during

"Test" mode, all Python printed output is presented in a popup window. An additional popup window presents Python error messages and error traceback.

After "Apply" is pressed, the Python error messages will appear in popup windows; however, the printed output will not be displayed. Instead it is diverted to the VAPOR diagnostics. Those printed messages can still be viewed in the VAPOR log file: From the Edit menu, click "Edit User Preferences" and set the value of "Log File Max" for Diagnostic Messages to a positive value, to ensure that these messages will appear in the log file.

Advanced usage:

Python is a powerful programming language and it can be used to do much more than applying simple formulas to existing variables. Python scripts can use data at other timesteps or coordinates to determine the value of an output variable at a specific time and position. For instance, such calculations are useful when calculating derivative operators or summarizing 3D information in a 2D variable . VAPOR users can apply existing Python libraries to calculate and visualize derived variables.

The Python environment used in deriving a VAPOR variable:

When a Python script is used in VAPOR to calculate an output variable, it is required that the script produce a NumPy (3D or 2D) array of 32-bit floating point (float32) values, with "C" ordering. All the variables in the VAPOR environment are represented as collections of 2D or 3D arrays, one array for each time step. The 2D arrays use the horizontal (X and Y) spatial dimensions of the VDC. The 3D arrays correspond to the three (X,Y,Z) spatial dimensions of the VDC, where Z is the vertical height. The dimension order of the array in Python is reversed from the dimensions used in VAPOR, due to the fact that VAPOR data arrays are accessed in FORTRAN order, while Python defaults to using C coordinate ordering. Thus, the first coordinate of a 3-dimensional array in the Python environment corresponds to the Z coordinate in the VAPOR visualization. The second Python coordinate is the Y coordinate in the VAPOR scene, and the third Python coordinate is the X coordinate in VAPOR. When dealing with two-dimensional variables, the first coordinate in the Python array is the Y coordinate in the VAPOR scene, and the second coordinate in the Python array is the X coordinate in VAPOR.

The vapor module.

VAPOR provides an internal module, "vapor" that is only visible if you are executing python scripts in the VAPOR application. This module provides access to the internal state of VAPOR, which is sometimes needed in deriving variables in the Python environment.

There are four predefined variables that are defined in the vapor module and may be needed for calculating a variable:

vapor.TIMESTEP : The integer time step of the output variable(s) being calculated

vapor.REFINEMENT : The integer refinement level of the output variables being calculated

vapor.LOD : The integer level-of-detail (compression level) that is being calculated.

vapor.BOUNDS : This is a 6-tuple that specifies the integer extents of the Z, Y, and X coordinates of the variable being calculated, inside the full 3D domain of the current VDC. These values may specify a region larger than the region being currently visualized in VAPOR, because VAPOR always stored variables in blocks (usually 32 or 64 voxels on a side). Using "B" to denote "vapor.BOUNDS", the output variable from a script, as a NumPy array, must have shape equal to:
[B[3]-B[0] +1, B[4]-B[1]+1, B[5]-B[2]+1] if it is 3-dimensional;
if it is 2-dimensional, the output NumPy array shape must be:
[B[4]-B[1]+1, B[5]-B[2]+1]

Note that the value of vapor.BOUNDS_is based on the extents of the output variable that is requested at runtime by the VAPOR visualization. When the output variable is 2D, the Z extents (from BOUNDS[0] to BOUNDS[3]) are set to the Z bounds of the full data domain.

Note also that, for efficient data access, VAPOR always stores variables on block boundaries, so the values of BOUNDS will often describe a volume of data that properly contains the current region in the VAPOR scene. Each element of BOUNDS will typically be a multiple of 32, except that the largest value in BOUNDS will be no greater than the largest value of the corresponding coordinate in the VDC.

There are also a few Python methods that are provided for use in accessing data in the VDC. These are also in the "vapor" module, which is already imported prior to script execution. They include:

userCoord = vapor.MapVoxToUser(voxcoord, refinementLevel [,lod]) : This returns a floating-point 3-tuple, "userCoord" that is obtained by converting the specified integer 3-tuple voxcoord, at the specified integer refinement level and compression level (lod). If lod is not specified, it defaults to 0.

voxCoord = vapor.MapUserToVox(usercoord, refinementLevel [,lod]) : This returns an integer 3-tuple, "voxCoord" that is obtained by converting the specified floating-point 3-tuple usercoord, at the specified integer refinement level and compression level (lod). If lod is not specified, it defaults to 0.

maxCoord = vapor.GetValidRegionMax(timestep,variableName, refinementLevel) : This returns an integer 3-tuple, "maxCoord" that is the maximum of the valid voxel coordinates of the variable at the specified timestep and refinement level.

minCoord = vapor.GetValidRegionMin(timestep,variableName, refinementLevel) : This returns an integer 3-tuple, "minCoord" that is the minimum valid voxel coordinates of the variable at the specified timestep and refinement level.

var3 = vapor.Get3DVariable(timestep, variablename, refinement, lod, bounds) : returns a 3-dimensional array of variable data from the VDC, based on:

timestep (integer time step)
variablename (the variable name, a string)
refinement (the refinement level, an integer)
lod (the level of detail or compression level, an integer)
bounds (a 6-tuple specify the location of the data in the full domain, as with vapor.BOUNDS described above.

var2 = vapor.Get2DVariable(timestep, variablename, refinement, lod, bounds) : returns a two-dimensional array of data from a 2-dimensional variable in the VDC, with arguments as above. The "bounds" argument must be an integer 6-tuple, however the first and fourth elements are ignored.

val = vapor.VariableExists(timestep, varname [refinement,[lod]]): returns 0 or 1 (False or True) depending on whether the specified variable is present at the specified timestep, refinement, and compression lod. If the refinement level or lod is not specified, the default value is 0.

msk = vapor.Get3DMask(timestep, varname ,refinement,lod, bounds) returns the boolean 3D mask variable indicating the location where the 3d variable "varname" is not set to its missing value.

msk = vapor.Get2DMask(timestep, varname ,refinement, lod, bounds) returns the boolean 2D mask variable indicating the location where the 2d variable "varname" is not set to its missing value.

Note that when you write your own Python script, the script will have access to all variables in the VDC (at the current refinement, bounds, timestep, and lod) that are specified in the "Input Variables" checkboxes. It is only necessary to use the above Get2Dvariable() and Get3Dvariable() methods when the VDC variables are needed at coordinates, refinement levels, levels-of-detail, or timesteps different from those of the output variables.

Missing Values. When the VDC contains variables with missing values (e.g. ROMS, POP, or MOM model outputs), the vapor python environment utilizes the numpy.ma

(masked array) module, so that array operations are only applied to valid data. This should be transparent to users unless user python routines access individual elements in the model variables. In that case, user routines may need to utilize the Get2DMask() and Get3DMask() in order to ensure that operations are only applied to valid data in the arrays. Note that the VAPOR Python environment does not support user defining of masks. The only masks that are supported are those defined by the VDC, which are set at data conversion time.

Using VAPOR with other Python modules

The installation of VAPOR 2.4.0 contains an installation of Python 2.7.8 with NumPy 1.8.1 and SciPy 0.14.0 in the python2.7 subdirectory of the VAPOR lib directory. The corresponding Python interpreter executable is installed in the VAPOR bin directory. VAPOR sets the environment variable PYTHONHOME inside the vaporgui application when any Python scripts are run. This setting will not persist and is not visible to other applications, so that other applications will not be affected by this Python version. Users can modify the default Python installation in (at least) two ways:

- Extend the VAPOR Python environment: Other useful Python modules, such as PyNIO, etc. can be installed inside the VAPOR python environment. Apply the VAPOR python executable to the setup.py script associated with the module. The module must be compatible with Python 2.7.

- Specify that VAPOR use a different Python 2.7 environment: VAPOR can be redirected to use another Python 2.7 environment in either of two ways:

    - Set the environment variable PYTHONHOME to the root directory of the other Python environment prior to starting vaporgui. On startup VAPOR will display a warning message indicating that the value of PYTHONHOME has been changed. It is required that NumPy be installed in the site-packages directory in your Python environment, and it is also necessary that the new version of Python be the same version of Python (2.7) with which VAPOR is linked.
    - Set the environment variable VAPOR_PYTHONHOME to the root directory of another Python 2.7 environment (which must have NumPy installed) before running vaporgui. This is a way to specify the Python environment that VAPOR uses without affecting other applications, and without modifying the value of PYTHONHOME.