In this post I will show how to 'convert' NumPy arrays to VTK arrays and files by means of the `vtk.util.numpy_support` module and the little-known PyEVTK package respectively.

Intro: The Conundrum

I wrote, or rather ranted, in my previous post about the value of VTK. Now lets say you were convinced (ha!) and decided to start including VTK in your scripts for visualization and processing.

Well, as if there weren't enough deterrents in employing VTK, you will quickly realize that using your precious data – which let's face it – will be stored in NumPy `ndarray` objects, with VTK ain't all that straightforward. And why would it be? VTK was made in C++ and C++ isn't about ease-of-use and concise programing. C++ is about putting hair on your chest :).

The traditional/ugly way, is creating new VTK objects, setting a bunch of properties like dimensions etc, and looping over your NumPy data to copy and populate your new objects. Since, looping in Python must be avoided like the black plague I will be focusing on the two ways I prefer.

The first way is using the `vtk.util.numpy_support` module that comes with VTK and allows you to 'easily' convert your data. The second way is by means of exporting your data into VTK-readable files using the PyEVTK package, a way which as you'll see is great if you want to process and/or visualize that data in VTK-based applications.

Using the `numpy_support` module

So, given the popularity of Python and the fact that VTK is exposed in its near entirety to Python, the VTK folk decided to create the `numpy_support` module which resides under `vtk.util`. Of course, given the near-absence of documentation and/or examples, using it is as convoluted as doing anything in VTK. However, I'm here to try and elucidate their usage.

Usage

The functions of interest to us are `numpy_to_vtk` and `vtk_to_numpy`. Let us first inspect the docstring of the first function which can be accessed as follows, assuming you have VTK installed in your Python distro:

```
from vtk.util import numpy_support
help(numpy_support.numpy_to_vtk)
```

with the result being

```
numpy_to_vtk(num_array, deep=0, array_type=None)
    Converts a contiguous real numpy Array to a VTK array object.

    This function only works for real arrays that are contiguous.
    Complex arrays are NOT handled.  It also works for multi-component
```

```
    arrays.  However, only 1, and 2 dimensional arrays are supported.
    This function is very efficient, so large arrays should not be a
    problem.

    If the second argument is set to 1, the array is deep-copied from
    from numpy. This is not as efficient as the default behavior
    (shallow copy) and uses more memory but detaches the two arrays
    such that the numpy array can be released.

    WARNING: You must maintain a reference to the passed numpy array, if
    the numpy data is gc'd and VTK will point to garbage which will in
    the best case give you a segfault.

    Parameters
    ----------

    - num_array :  a contiguous 1D or 2D, real numpy array.
```

Upon first inspection, one might think that 3D NumPy arrays weren't possible to convert. At least that's what I thought (yeah, yeah, I suck). However, all that one needs to do is create a 1D representation of the array using 'numpy.ndarray' methods such as `flatten` or `ravel`. So here's how to use this function assuming you mean to export an `numpy.ndarray` object named `NumPy_data` of type `float32` :

```
NumPy_data_shape = NumPy_data.shape
VTK_data = numpy_support.numpy_to_vtk(num_array=NumPy_data.ravel(), deep=True, array_type=vtk.
```

As you can see we use `ravel` to flatten `NumPy_data` using the default C, or row-major, ordering (especially important on 2D and 3D arrays). In addition, we specify that we **do** want the data to be deep-copied by setting `deep=True`, while we also define the data type by `array_type=vtk.VTK_FLOAT`. Note, that we keep a copy of the `shape` of `NumPy_data` which we will use later to `reshape` the result of `vtk_to_numpy`. Converting back is much easier and can be done as such:

```
NumPy_data = numpy_support.vtk_to_numpy(VTK_data)
NumPy_data = NumPy_data.reshape(NumPy_data_shape)
```

> CAUTION: You may choose to allow for shallow-copies by setting `deep=False` but be warned: If for any reason, the array you pass is garbage-collected then the link will break and your nice VTK array will be useless. Should that occur, if you end up using `vtk_to_numpy` to back-convert, you will simply get memory trash. That is especially the case if you use `flatten` instead of `ravel` as the former always returns a 'flattened' copy of the array (check docs here) which is bound to get gc'ed if you use it directly in the call. `ravel` is slightly safer as it typically returns a new 'view' to the array and only copies when it has to (check docs here) but you don't want to depend on that. Bottom line: disable deep-copying at your own risk and only if you know what you're doing.

Using the PyEVTK package

Some time ago, I was struggling with the `numpy_support` module discussed above, mostly cause I sucked and didn't think to `ravel` the array, so I started googling and came across PyEVTK a great little package by a Paulo Herrera. While back then, the code relied on C routines which refused to work on Windows platforms, since v0.9 – v1.0.0 being the current version – the package is built on pure-Python code and works like a charm.

What `PyEVTK` does, is allow you to save NumPy arrays straight to different types of VTK XML-based file formats (not the old legacy ones), without even needing to have VTK installed in your system, making NumPy the only dependency. This way, you can easily save your NumPy arrays into files that can be visualized and processed with any of the flagship VTK applications such as ParaView, VisIt, Mayavi, while obviously you can easily load said files with VTK and use all the goodies the toolkit offers.

However, while I'm supremely grateful to Mr. Herrera for creating PyEVTK, the package wasn't hosted on PyPI and could only be used by checking out the code from the original repository in BitBucket, and using `distutils` to build/install it via the good ol' fashioned `python setup.py install`.

Thus, possessed by the noble spirit of plagiarism, I took it upon myself to rip off, i.e., fork, the repository, re-package it, and upload it on PyPI. You can now find my fork on BitBucket here, while the package is hosted on PyPI under `PyEVTK` here. Therefore, you can install with `pip` with the following command:

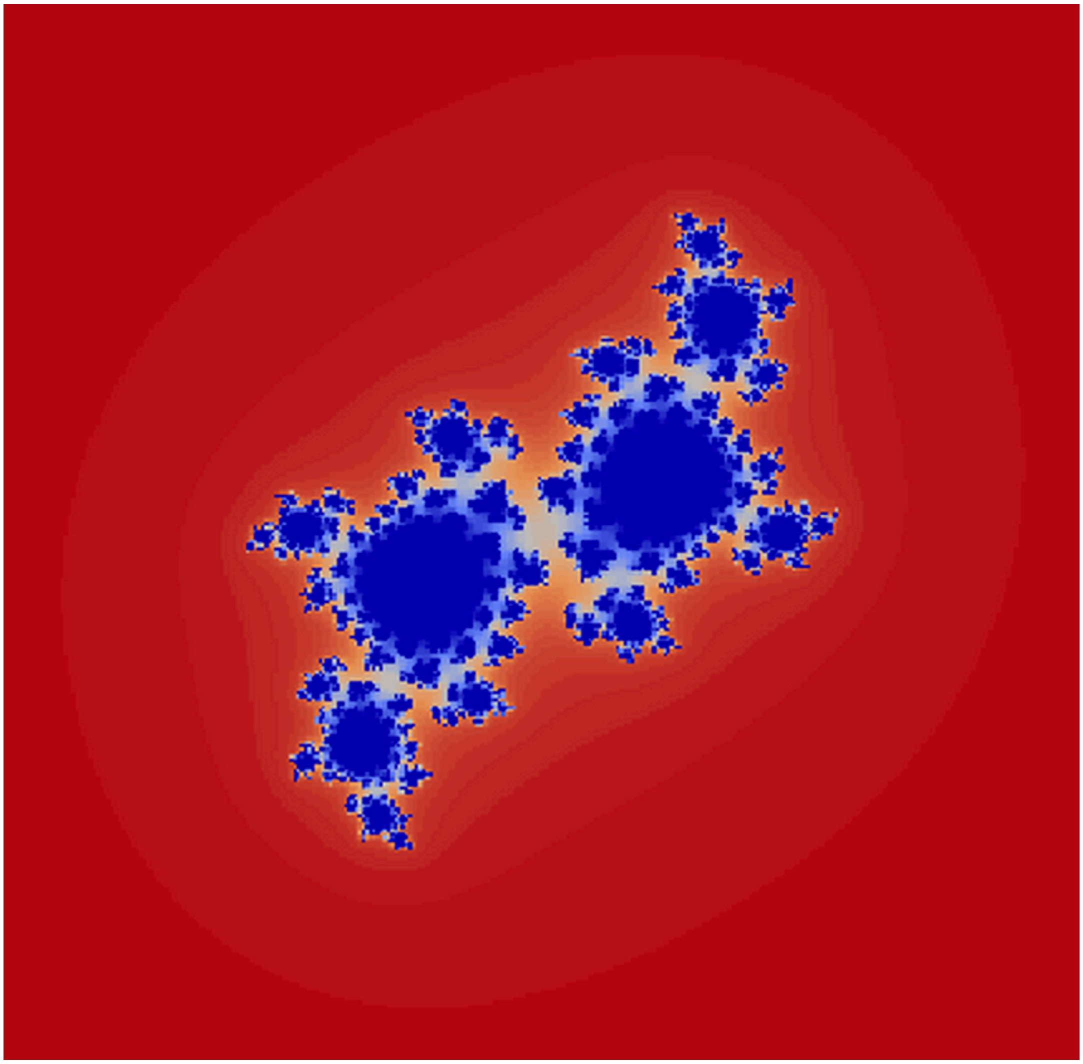
```
pip install pyevtk
```

> As you'll see on PyPI, I claim no ownership, authorship, or entitlement of PyEVTK. Paulo Herrera wrote the thing and the only reason I 'stole' it was cause I use it a lot and wanted to make it widely available.

Usage

I'm gonna give a quick example here to show you how to save a NumPy array as a rectilinear-grid file with a `.vtr` extension. In the interest of consistently misappropriating other people's code, I modified the code for creating a Julia set from Ted Burke's post here. I just thought I'd use a pretty dataset for my demo :)

Here, however, I'm only going to show the PyEVTK part of the code while you can see the full thing under this IPython Notebook. The result of the Julia code is the following pretty 2D array which was visualized with Plotly's `Heatmap` method (see this previous post on how to use Plotly):



A Julia set visualized with Plotly

So, here's the PyEVTK part of the notebook:

```
from pyevtk.hl import gridToVTK

noSlices = 5
juliaStacked = numpy.dstack([julia]*noSlices)

x = numpy.arange(0, w+1)
y = numpy.arange(0, h+1)
z = numpy.arange(0, noSlices+1)

gridToVTK("./julia", x, y, z, cellData = {'julia': juliaStacked})
```

The first important part of this code is `from pyevtk.hl import gridToVTK` which uses the high-level functions of the `pyevtk` package and which reside under the `pyevtk.hl` module. For the most-part, this module has all you need but the package also provides a `pyevtk.vtk` module for low-level calls. As we want to export data residing on a rectilinear grid we use the `gridToVTK` method.

The next 5 lines aren't important but what we do is use the `numpy.dstack` function to stack 5 copies of the 2D `julia` array and create a 3D array to export. Note the `numpy.arange` calls through which we calculate 'fake' axes for the array to be exported. An important point to make is that since we're defining a grid we need axes with N+1 coordinates which define the grid edges (as the data resides in the centers of that grid). If you want to export point-data, use the `pyevtk.hl.pointsToVTK` function.

As you guessed the magic happens with this one line:

```
gridToVTK("./julia", x, y, z, cellData = {'julia': juliaStacked})
```

As you can see, we pass the path to the file **without an extension** which will be automatically defined by PyEVTK depending on which function you used in order to create the appropriate VTK file. Subsequently, we pass the three coordinate axes, and provide a Python dictionary with a name for the array and the array itself. Doesn't get easier than that :). The result of the above is `.vtr` file which you can find here while the IPython Notebook of the entire code can be found here.

Resources

A quick note here: while Paulo's repo was named PyEVTK, he had originally named the package `evtk` but in the interest of consistency I renamed it to `pyevtk` so his examples won't work directly if you installed PyEVTK through `pip` (see above section). If you want to see more examples of its usage, I refactored them a tad and can be seen in my repo here.

Well that's it for today. Thanks for reading once more and I hope you gained something from this. Happy Python-ing :D