Py Notes

Applications and examples...

Published - 2016-04-19
Dan Patterson

Distance... distance... distance...

One point to one point. One point to many points. Many to many. Line length, polygon perimeter.

Single-part, multi-part. 1D, 2D or 3D what about 4D or more?

So many things to consider, so many ways to do the calculations.

Which is the fastest? Which is the coolest?

Well I have blogged on this before and I am sure I will again. This note is to demonstrate some simple usage of **einsum** notation. Yes, Einstein is at the root of this... circa 1916.

I am not even going to attempt to explain how it works in detail, since my interest is focused specifically on how to calculate distance between objects and/or the measurement of length and its terminological variants. In short...the syntax is short and sweet. Complex operations that involve products and summations are handled cleverly and quickly... not the fastest... perhaps, but for GIS practical solutions...more than fast enough.

The code is listed on the next. It boils down to two simple functions. The first, **e_dist,** is used to determine the euclidean distance between one point and another, or an array of other points. The points can be 2D (x,y) or 3D (x,y,z). The origin point could obviously be any point, or pulled out of a single array or from another array. The second function, **e_leng,** is used to determine distances between many points for the calculation of length or perimeter.

I will leave the discussion of the 'magic' behind einsum for my **Esoterica** documents.

I have placed the two salient lines of code in larger bold-faced font. In boils down to determine the difference in coordinates between two point or point sequences and applying the summation of the products of the arranged terms. The rest of the code is used to formulate the terms of the inputs depending upon whether distance is being calculated between:

- two single points
- one point to a sequence of points
- a sequence of points to another sequence of points
- and the above can be using 2D or 3D planar coordinates

```
import numpy as np
from textwrap import dedent
np.set_printoptions(edgeitems=4,linewidth=90, precision=3,
                    suppress=True, threshold=10)

def e_dist(a, b, keepdims=False):
    """Distance calculation for 2D and 3D points"""
    a_dim = a.ndim; b_dim = b.ndim
    b = np.atleast_2d(b)
    if a_dim == 1:
        #origs = np.atleast_2d(origs)
        a = a.reshape(1,1,a.shape[0]) #
    if a_dim == 2:
        r,c = a.shape
        a = np.reshape(a, (a.shape[0],1,a.shape[1]))
    if b_dim > 2:  # when destinations are 3 or more
        s = b.shape
        b = b.reshape(np.prod(s[:-1]),s[-1])
    #
    diff =  a - b
    dist_arr = np.sqrt(np.einsum('ijk,ijk->ij', diff, diff))
    #
    if (b_dim > 2) and keepdims:
        dist_arr = dist_arr.reshape(a.shape[0],-1,b.shape[-1])
    dist_arr = np.squeeze(dist_arr)
    return dist_arr
# ----
# Sample data for all...
pnt = np.array([[0.,0.]])
origs = np.array([[0.,0.],[1.,0.],[0.,1.],[1.,1.]])
dests = np.asarray([[4.,0.],[0.,2.],[2.,2.],[-2.,-3.],[4.,4.]])
#
```

Single origin to multiple destinations

Example 1 shows how to calculate a single origin to multiple destination points. It cleverly boils down to the one-liner calling **e_dist** using a single input point and multiple destinations.

If your intent is to find closest points, you can omit the square root portion in the function since the squared distance will yield the same results as those fully calculated.

```
# ---- Example 1
# use e_dist for distances
#
>>> p = np.array([0.,0.])
>>> d = np.asarray([[4.,0.],[0.,2.],[2.,2.],[-2.,-3.],[4.,4.]])
>>> e_dist(p, d, keepdims=True)
array([ 4.   ,  2.   ,  2.828,  3.606,  5.657])
```

It should be obvious that the output array shows the distances from the origin to all the destinations.

Multiple origins to multiple destinations

The only magic in the second example is the reshaping of the origin array so that each entry can be used in the subtraction and distance calculation. This in effect is just a generalization of the previous function.

I used a couple of output options to help with the understanding and the numbers are simple enough so that the calculations can be verified.

The results are as follows.

```
# ---- Example 2
#  reshape the array so that every x,y pair is treated
#  separately. origin array: origs.shape = (4,2) using,
#  origs.reshape(4,1,2): or origs[:,None,:]

>>> o = np.array([[0.,0.],[1.,0.],[0.,1.],[1.,1.]])
>>> d = np.asarray([[4.,0.],[0.,2.],[2.,2.],[-2.,-3.],[4.,4.]])
>>>
>>> e_dist(o, d, keepdims=True)
array([[ 4.   ,  2.   ,  2.828,  3.606,  5.657],
       [ 3.   ,  2.236,  2.236,  4.243,  5.   ],
       [ 4.123,  1.   ,  2.236,  4.472,  5.   ],
       [ 3.162,  1.414,  1.414,  5.   ,  4.243]])
```

The output format for the above is on a origin point to destination array basis. The first row in the output is obviously the same as when we used the (0,0) origin cast against all the destinations in the destination array. Subsequent rows are the results for different origins against the destinations. The data can also be read column-wise should you be interested in that view without having to switch the input orders.

Of course, the examples shown give simple command line outputs, but I have included helper functions that produce fancier outputs that describe he forms of the inputs and outputs in more detail.

Length calculation from point sequences

The calculation of length is carried out in a similar fashion but the differences in the coordinate values are obviously sequential. If a feature contains multiple parts, the lengths/perimeter is determined for each part.

The results of the function return the inter-point distances and the total length/distance that they represent.

```
def e_leng(a, verbose=True):
    """edge lengths using einsum
       Inputs:  a list/array coordinate pairs, with ndim = 3 and the
       minimum shape = (1,2,2), for example, (1,4,2) for a single line of 4 pairs
       The minimum input needed is a pair, but a sequence of pairs can be used.
       """
    #a = np.asanyarray(a, dtype='float64')
    if a.ndim == 2:
        r,c = a.shape
        a = np.reshape(a, (1,r,c))
     #
     diff = a[:,0:-1] - a[:,1:]
     dist = np.sqrt(np.einsum('ijk,ijk->ij', diff, diff))
     length = np.sum(dist, axis=1)
     #
     if verbose:
        frmt = """
        Input array....(shape={1})
        {0}\n
        differences...
        {2}\n
        distances...
        {3}
        length...{4}"""
        print(dedent(frmt).format(a,a.shape,diff,dist,length))
    return dist, length
```