More Raster Processing
(or there is more than one way to skin a cat)

Open Source RS/GIS Python
Week 6
Projecting rasters

- Need Well Known Text (WKT) for input and output projections
- Can get it from the original Dataset (if it has a projection defined) with `GetProjection()`
- Can create output WKT using the `SpatialReference` objects we learned about earlier
• **gdal.CreateAndReprojectImage**
  ```python
  <source_dataset>, <output_filename>,
  src_wkt=<source_wkt>,
  dst_wkt=<output_wkt>,
  dst_driver=<Driver>,
  eResampleAlg=<GDALResampleAlg>)
  ```

• There are a few other options that I won't cover here

• Sets geotransform and projection but does not build pyramids
import gdal, osr
from gdalconst import *

inFn = 'd:/data/classes/python/data/aster.img'
outFn = 'd:/data/classes/python/data/aster_geo.img'

driver = gdal.GetDriverByName('HFA')
driver.Register()

# input WKT
inDs = gdal.Open(inFn)
inWkt = inDs.GetProjection()

# output WKT
outSr = osr.SpatialReference()
outSr.ImportFromEPSG(4326)
outWkt = outSr.ExportToWkt()

# reproject
gdal.CreateAndReprojectImage(inDs, outFn, src_wkt=inWkt,
   dst_wkt=outWkt, dst_driver=driver,
   eResampleAlg=GRA_Bilinear)

inDs = None
Method comparison

• Simple model using a DEM
  • elevation > 2500 = 1
  • elevation <= 2500 = 0
  • Small DEM (1051 X 1397)
Pixel by pixel processing

- Can loop through each pixel with Numeric

```python
outData = Numeric.zeros((rows, cols))
for y in range(rows):
    for x in range(cols):
        if inData[y, x] > 2500:
            outData[y, x] = 1
        else:
            outData[y, x] = 0
```
Built-in function

• Or can use a built-in Numeric (or numpy) function whenever possible

\[ \text{outData} = \text{numpy.greater(inData, 2500)} \]
Another comparison

```
if elevation > 2000:
    if awch > 0.15: output = 1
    else: output = 0
else:
    if awch > 0.2: output = 1
    else: output = 0
```

Elevation

Soil available water capacity (awch)

Output
Pixel by pixel

```python
outData = Numeric.zeros((rows, cols), Numeric.Int)
    for y in range(rows):
        for x in range(cols):
            if elev[y, x] > 2000:
                if awch[y, x] > 0.15:
                    outData[y, x] = 1
                else:
                    outData[y, x] = 0
            else:
                if awch[y, x] > 0.2:
                    outData[y, x] = 1
                else:
                    outData[y, x] = 0
```
Built-in functions

• Method 1
  
case1 = Numeric.where((elev > 2000) & (awch > 0.15), 1, 0)
case2 = Numeric.where((elev <= 2000) & (awch > 0.2), 1, 0)
outData = case1 + case2

• Method 2
  
case1 = Numeric.where(Numeric.greater(elev, 2000) &
      Numeric.greater(awch, 0.15), 1, 0)
case2 = Numeric.where(Numeric.less_equal(elev, 2000) &
      Numeric.greater(awch, 0.2), 1, 0)
outData = case1 + case2
• Method 3

```python
outData = Numeric.where(Numeric.greater(elev, 2000),
    Numeric.where(Numeric.greater(awch, 0.15), 1, 0),
    Numeric.where(Numeric.greater(awch, 0.2), 1, 0))
```
## Results

<table>
<thead>
<tr>
<th>Method</th>
<th>My old PC (Numeric)</th>
<th>Numeric on Windows VM</th>
<th>Numpy on Windows VM</th>
<th>Numpy on Mac</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEM</td>
<td>28.4</td>
<td>6.5</td>
<td>7.5</td>
<td>12.9</td>
</tr>
<tr>
<td>DEM</td>
<td>0.14</td>
<td>0.06</td>
<td>0.0</td>
<td>0.005</td>
</tr>
<tr>
<td>Tree</td>
<td>n/a</td>
<td>39.9</td>
<td>49.3</td>
<td>77.1</td>
</tr>
<tr>
<td>Tree</td>
<td>n/a</td>
<td>1.5, 1.6, 1.3</td>
<td>0.7, 0.7, 0.7</td>
<td>0.4, 0.4, 0.4</td>
</tr>
</tbody>
</table>

Processing times in seconds

- Built-in functions are much faster than looping through pixels
Moving windows (neighborhoods)

- Neighborhood notation
- 3x3 average:
  \[ J = \frac{(E + F + G + I + J + K + M + N + O)}{9} \]
  - \( \text{out}[i,j] = \frac{(\text{in}[i-1,j-1] + \text{in}[i-1,j] + \text{in}[i-1,j+1] + \text{in}[i,j-1] + \text{in}[i,j] + \text{in}[i,j+1] + \text{in}[i+1,j-1] + \text{in}[i+1,j] + \text{in}[i+1,j+1])}{9} \)
• For 3x3 window, the output is 2 columns and 2 rows smaller than input
3x3 average pixel by pixel

• Write output to a band of type Byte
• Truncating the average (86.7 -> 86)
• Average gets truncated to integer when put into outData, which is type Int

```python
data = inBand.ReadAsArray(0, 0, cols, rows).astype(Numeric.Int)
outData = Numeric.zeros((rows, cols), Numeric.Int)
for i in range(1, rows-1):  # skipping first & last
    for j in range(1, cols-1):
        outData[i,j] = (data[i-1,j-1] + data[i-1,j] + data[i-1,j+1] +
                        data[i,j-1] + data[i,j] + data[i,j+1] +
                        data[i+1,j-1] + data[i+1,j] + data[i+1,j+1]) / 9.0
```
• Explicitly rounding the average (86.7 -> 87)
• Average gets rounded before being put into outData

data = inBand.ReadAsArray(0, 0, cols, rows).astype(Numeric.Int)
outData = Numeric.zeros((rows, cols), Numeric.Int)
for i in range(1, rows-1):  # skipping first & last
    for j in range(1, cols-1):
        outData[i,j] = round((data[i-1,j-1] + data[i-1,j] +
                              data[i-1,j+1] + data[i,j-1] + data[i,j] + data[i,j+1] +
                              data[i+1,j-1] + data[i+1,j] + data[i+1,j+1]) / 9.0)
• Implicitly rounding the average (86.7 -> 87)
• Average stays a float when put into outData (type Float) but rounding to Byte when written to output band (type Byte)

data = inBand.ReadAsArray(0, 0, cols, rows).astype(Numeric.Int)
outData = Numeric.zeros((rows, cols), Numeric.Float)
for i in range(1, rows-1): # skipping first & last
    for j in range(1, cols-1):
        outData[i,j] = (data[i-1,j-1] + data[i-1,j] + data[i-1,j+1] +
                        data[i,j-1] + data[i,j] + data[i,j+1] +
                        data[i+1,j-1] + data[i+1,j] + data[i+1,j+1]) / 9.0
3x3 average with array slices

• Basically slicing and shifting arrays
• Perform calculations on entire arrays rather than individual pixels
• Output and all input array slices MUST be the same dimensions
• Output array cannot be a smaller data type than any of the input arrays
• Substitute a reference to an array slice for a specific pixel
• Hatched areas are the i,j pixels that will get output values
• Shaded areas are the slices that go into the calculation

Pixel notation: data[i,j]
Slice notation: data[1:rows-1,1:cols-1]
Pixel: data[i-1, j-1]
Slice: data[0:rows-2, 0:cols-2]

Pixel: data[i, j+1]
Slice: data[1:rows-1, 2:cols]

Pixel: data[i+1, j]
Slice: data[2:rows, 1:cols-1]
• Truncating the average (86.7 -> 86)
• Because outData is Int, must keep everything integer during calculations (divide by 9 instead of 9.0)

```python
data = inBand.ReadAsArray(0, 0, cols, rows).astype(Numeric.Int)
outData = Numeric.zeros((rows, cols), Numeric.Int)
outData[1:rows-1,1:cols-1] = (data[0:rows-2, 0:cols-2] +
data[0:rows-2,1:cols-1] + data[0:rows-2,2:cols] +
data[1:rows-1, 0:cols-2] + data[1:rows-1,1:cols-1] +
data[1:rows-1,2:cols] + data[2:rows,0:cols-2] +
```
Explicitly rounding the average (86.7 -> 87)

Average gets rounded and then converted back to integer so it can be put into outData (type Int)

data = inBand.ReadAsArray(0, 0, cols, rows).astype(Numeric.Int)
outData = Numeric.zeros((rows, cols), Numeric.Int)
outData[1:rows-1,1:cols-1] = Numeric.around((
data[0:rows-2, 0:cols-2] + data[0:rows-2,1:cols-1] +
data[0:rows-2,2:cols] + data[1:rows-1, 0:cols-2] +
data[1:rows-1,1:cols-1] + data[1:rows-1,2:cols] +
• Implicitly rounding the average (86.7 -> 87)
• Average stays a float when put into outData (type Float) but rounding to Byte when written to output band (type Byte)

```python
data = inBand.ReadAsArray(0, 0, cols, rows).astype(Numeric.Int)
outData = Numeric.zeros((rows, cols), Numeric.Float)
outData[1:rows-1, 1:cols-1] = (data[0:rows-2, 0:cols-2] +
data[0:rows-2,1:cols-1] + data[0:rows-2,2:cols] +
data[1:rows-1, 0:cols-2] + data[1:rows-1,1:cols-1] +
data[1:rows-1,2:cols] + data[2:rows,0:cols-2] +
```
## Results

<table>
<thead>
<tr>
<th>Method</th>
<th>My old PC (Numeric)</th>
<th>Numeric on Windows VM</th>
<th>Numpy on Windows VM</th>
<th>Numpy on Mac</th>
</tr>
</thead>
<tbody>
<tr>
<td>Truncating pixel by pixel</td>
<td>44.3</td>
<td>13.5</td>
<td>34.7</td>
<td>44.2</td>
</tr>
<tr>
<td>Explicit round pixel by pixel</td>
<td>50.9</td>
<td>14.5</td>
<td>37.0</td>
<td>47.8</td>
</tr>
<tr>
<td>Implicit round pixel by pixel</td>
<td>47.4</td>
<td>13.1</td>
<td>22.5</td>
<td>23.7</td>
</tr>
<tr>
<td>Truncating slices</td>
<td>0.6</td>
<td>0.5</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>Explicit round slices</td>
<td>3.6</td>
<td>2.4</td>
<td>0.7</td>
<td>0.3</td>
</tr>
<tr>
<td>Implicit round slices</td>
<td>2.1</td>
<td>0.7</td>
<td>0.6</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Processing times in seconds

OS Python week 6: More raster processing [24]
Another way to get average

• To compute a 3x3 average we added 9 pixel values and divided by 9
  \[(p_{1}+p_{2}+p_{3}+p_{4}+p_{5}+p_{6}+p_{7}+p_{8}+p_{9}) \div 9\]
• Since \(1/9 = 0.111\), this is the same as
  \[(p_{1}+p_{2}+p_{3}+p_{4}+p_{5}+p_{6}+p_{7}+p_{8}+p_{9}) \times 0.111\]
• Which is the same as
  \[0.111p_{1}+0.111p_{2}+0.111p_{3}+0.111p_{4}+0.111p_{5}+0.111p_{6}+0.111p_{7}+0.111p_{8}+0.111p_{9}\]
Filters

- Low-pass filter
  - Used to smooth data
  - Every pixel is weighted the same – same as our 3x3 average

- High-pass filter
  - Used to enhance edges
  - Pixels have different weights

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.111</td>
<td>0.111</td>
<td>0.111</td>
</tr>
<tr>
<td>0.111</td>
<td>0.111</td>
<td>0.111</td>
</tr>
<tr>
<td>0.111</td>
<td>0.111</td>
<td>0.111</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.7</td>
<td>-1.0</td>
<td>-0.7</td>
</tr>
<tr>
<td>-1.0</td>
<td>6.8</td>
<td>-1.0</td>
</tr>
<tr>
<td>-0.7</td>
<td>-1.0</td>
<td>-0.7</td>
</tr>
</tbody>
</table>
Assignment 6a

- Use a 3x3 high pass filter to detect edges in band 1 of smallaster.img
- The output data type will be Float
- Use pixel notation (that’s why you’re doing it on smallaster.img instead of aster.img)
- Turn in your code and a screenshot of the output
Assignment 6b

- Use a 3x3 high pass filter to detect edges in band 1 of aster.img (good idea to test on smallaster.img first)
- The output data type will be Float
- Use slice notation
- Turn in your code and a screenshot of the output
• Compare your output to output.img (it’s a subset of smallaster.img)

• No class next week