Open source geospatial programming in Python

A crash course..

William Armstrong
william.armstrong@colorado.edu

Geological Sciences
Institute for Arctic and Alpine Research
University of Colorado at Boulder

September 20, 2016

Today's goals

By the end of today's workshop, you will understand several fundamental Python concepts and be able to *programatically*[1]

- Read, manipulate, and write shapefiles and raster data
- Perform simple analyses on geospatial data
- Plot data derived from geospatial sources

[1] Through program code, rather than a user interface – wictionary.org

Why geospatial Python?

- Provides a multitude of options for handling geospatial data analysis and display
- Allows easy automation of tasks and manipulation & analysis of large datasets
- Runs on Mac's (and Windows)..
- Free and open

My motivation

Turning pretty pictures into quantitative data.

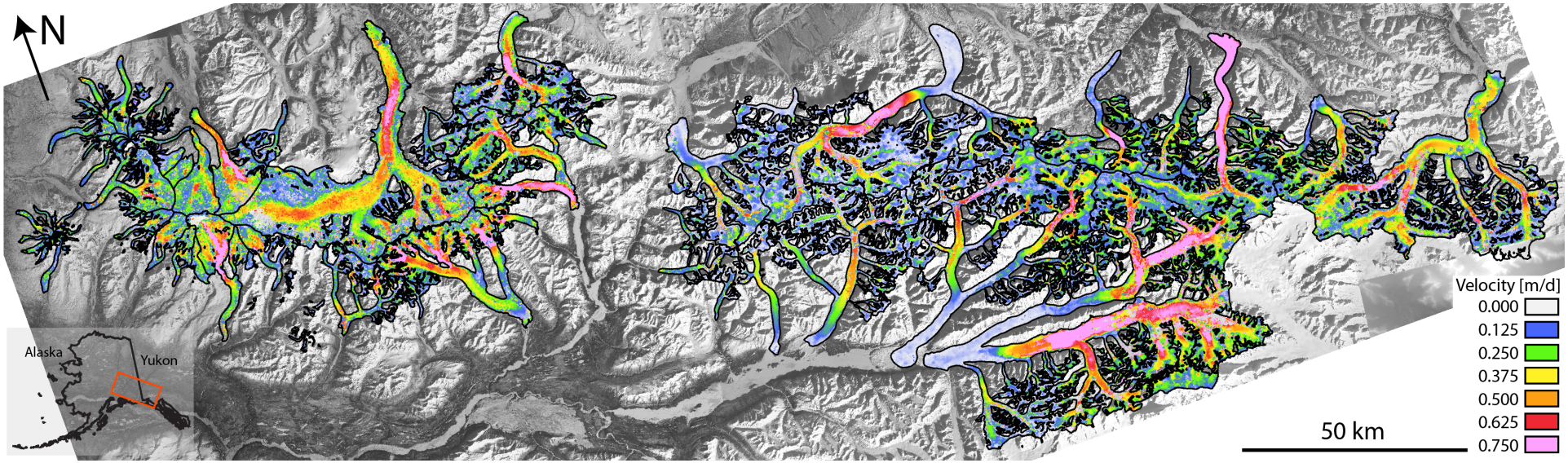


Figure: Glacier velocity on a range-wide scale over south-central Alaska.

My motivation

Repeat analysis of satellite imagery to inform our understanding of glacier dynamics.

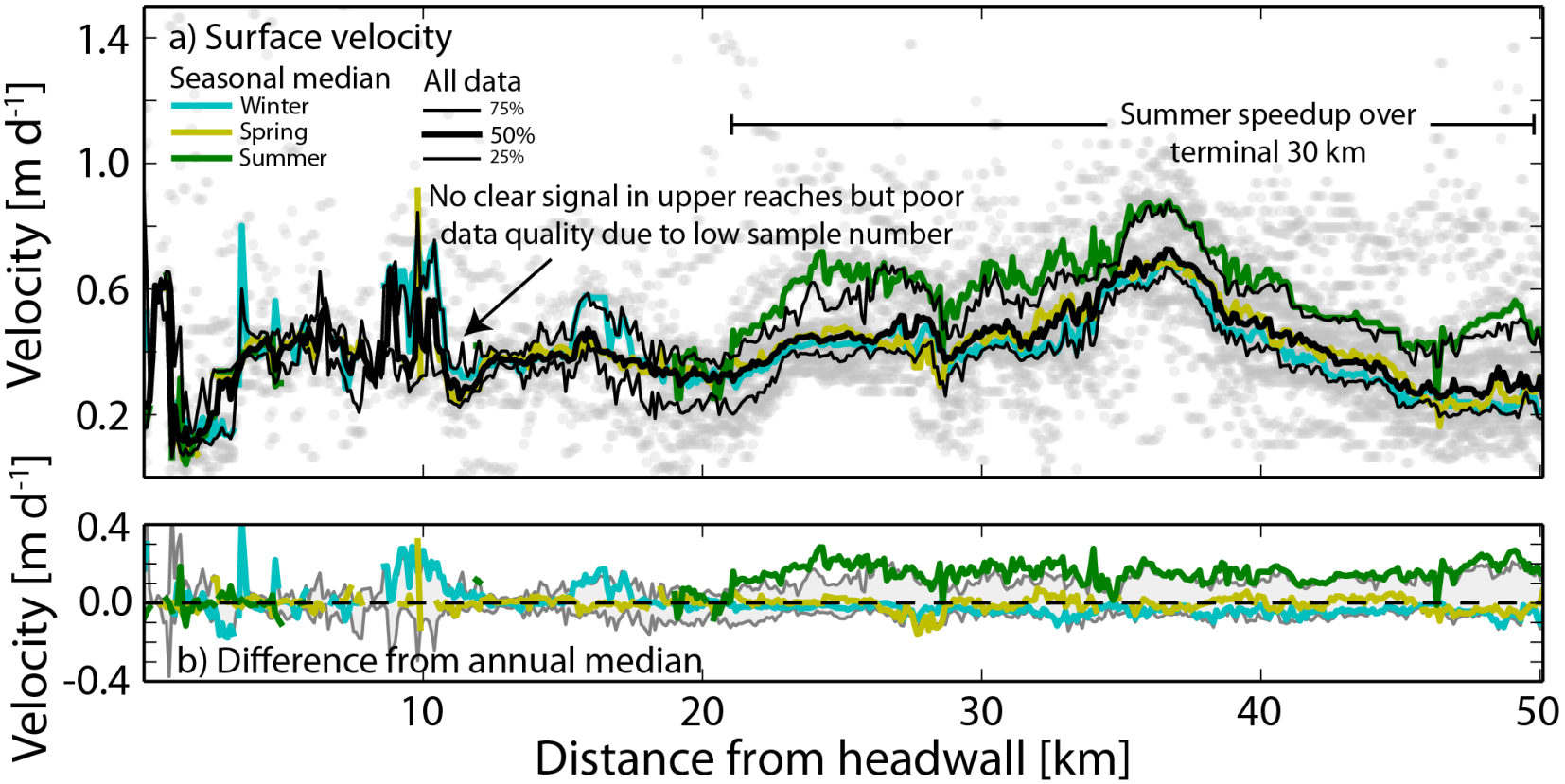


Figure: Seasonal glacier velocity along a down-glacier profile.

Python basics: indexing data

Different data structures have different ways of accessing contained values.

- **Tuples**: Square brackets with [row][column] indexing. Start = 0.

```
t = ( (1,2) , (5,9) ) , t[0][1] = 9
```

- **Lists**: Square brackets with [order] indexing.

```
l = [ 1, 2, 5, 9], l[2] = 5
```

- **Dictionaries**: Square brackets with ['key'] indexing.

```
d = { 'age':( (33,31,19) ), \
'name': ['john','claire','steve'] }
d['age'] = (33,31,19), d['age'][0] = 33
```

Python basics: for loops

A **for loop** is a way to get Python to repeat a specified sequence of tasks many times. Python's for loop structure is very particular about indenting, unlike some other languages. A Python for loop looks like:

```
1 a = [ 13, 5, 2, 4, 9, 16]
2 for i in range(0,len(a)):
3         print "Value " + i + " is " + a[i]
4 print "Done with loop"
```

Figure: Around and around we go. Image from wikipedia.org

Python basics: importing packages

Part of what allows Python to be so lightweight is that is only loads the tools you ask for. These tools are called "packages"; today we will be using:

```
# load a module and give it a short-hand
import matplotlib.pyplot as plt
import numpy as np
# load one module from a package
from shapely import geometry
# load a whole package
import fiona
import rasterio
```

Geospatial datatypes

Python can readily handle geospatial data. Geospatial data comes in two basic classes:

Raster: A representation of the world as a surface divided into a regular grid of cells. E.g., satellite imagery, DEMs. *GDAL, Rasterio*.

Vector: A representation of the world using points, lines, and polygons. E.g., roads, study area boundaries. *OGR, Shapely, Fiona*.

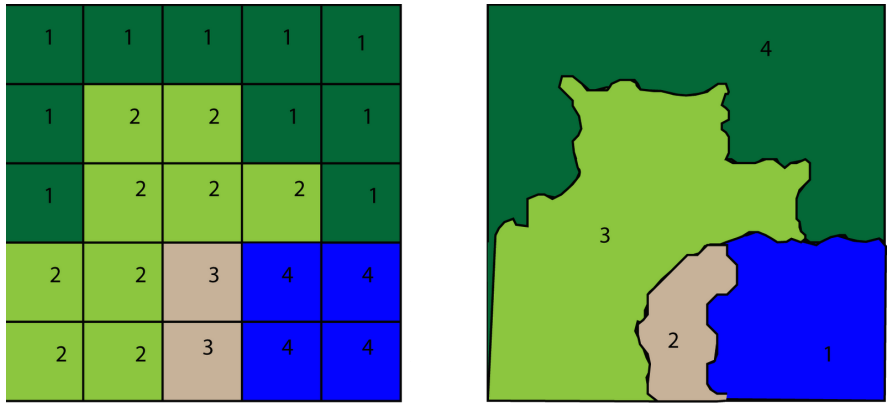


Figure: Raster (left) and vector (right) representations of a landscape.

Today's dataset

State shapefiles from
https://www.census.gov/geo/maps-data/data/cbf/cbf_state.html

First hit when Google "us state outlines shapefile" $\rightarrow$ "nation-based files" $\rightarrow$ "state" $\rightarrow$ "cb_2015_us_state_20m.zip"

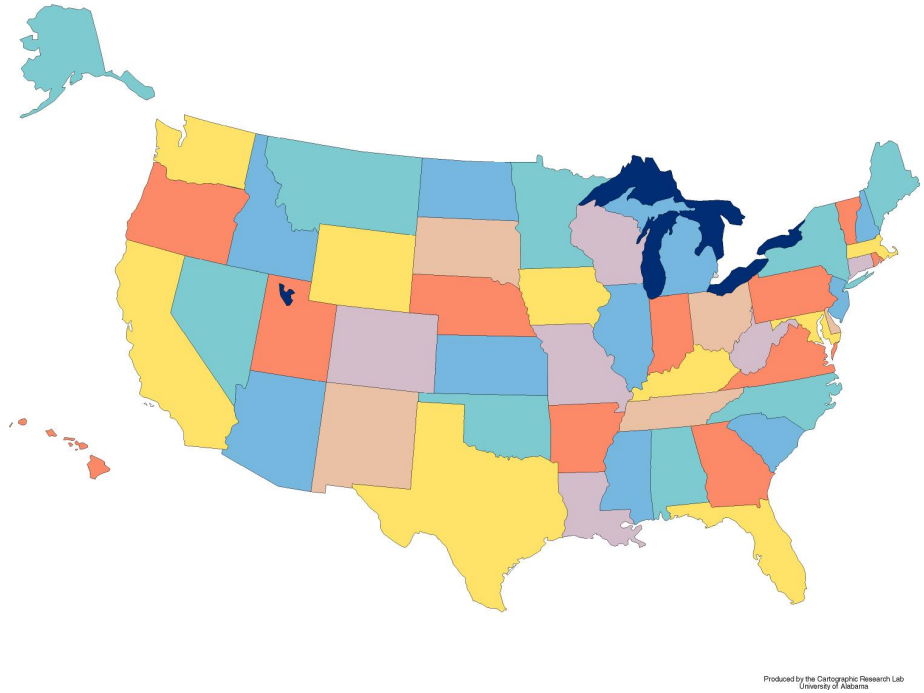


Image: alabamamaps.ua.edu

Reading vector data: Fiona

Fiona and Shapely are lightweight wrappers for the underlying OGR module, which is pretty clunky and not 'Pythonic'. Reading a shapefile with Fiona is pretty easy, as shown below.

```
1  stateFn = fullPath + 'cb_2015_us_state_20m.shp'
2  # open collection stateFn with 'r' for 'read'
3  states = fiona.open(stateFn,'r')
4
5  # Some basic information
6  len(states) # number of features within collection
7  states.crs # collection coordinate reference system (CRS)
8  states.schema # Information on fields and geometry type
9  state = states[0] # get the first feature and name it 'state'
10
11 # loop over features, print state names and USPS abbrev.
12 for state in states:
13         print state['properties']['NAME'], \
14               state['properties']['STUSPS']
```

Manipulating vector data: Shapely

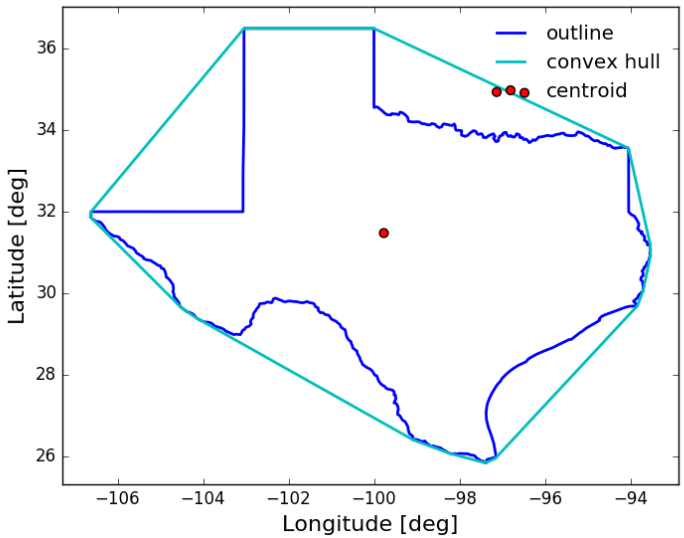
Fiona reads shapefiles, but Shapely is where the real geospatial power lies. Creating a Shapely geometry from a Fiona input looks like:

```
state = states[0] # get the first feature and name it 'state'
state.viewkeys() # ['geometry', 'type', 'id', 'properties']
shape = geometry.shape(state['geometry']) # make shapely geom
ext = shape.exterior # get the 'exterior' ring geom
x,y = ext.xy # 'exterior' ring coords
cent = ext.centroid # geometric centroid geom
cx,cy = cent.xy # centroid coords
hull = shape.convex_hull # convex hull geom
hx,hy = hull.exterior.xy # convex hull coords
```

See http://toblerity.org/shapely/manual.html for the many object attributes.

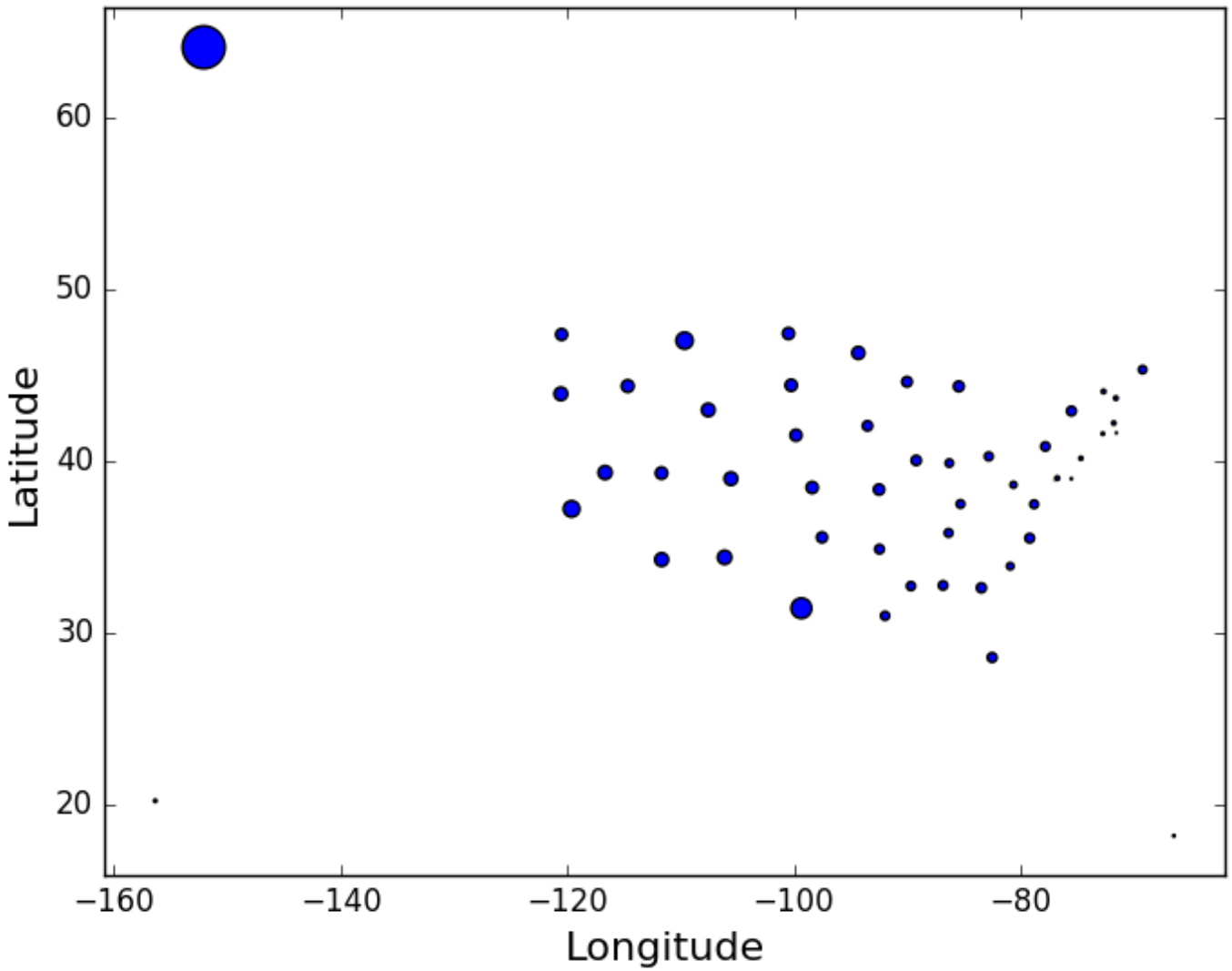
Plotting data: Matplotlib

```
plt.plot(x,y,c='b',lw=2,label='outline') # c = color, lw = linewidth
plt.plot(hx,hy,c='c',lw=2,label='convex hull') # label on legend
plt.scatter(cx,cy,s=40,marker='o',facecolor='r',label='centroid')
plt.xlabel('Longitude [deg]',fontsize=16) # ^s = size
plt.ylabel('Latitude [deg]',fontsize=16)
plt.legend(loc=1,frameon=False) # loc 1 = upper right corner
plt.axis('tight') # less white space
plt.savefig('texas_shapely.png',r=300) # r = resolution
plt.show()
plt.close()
```



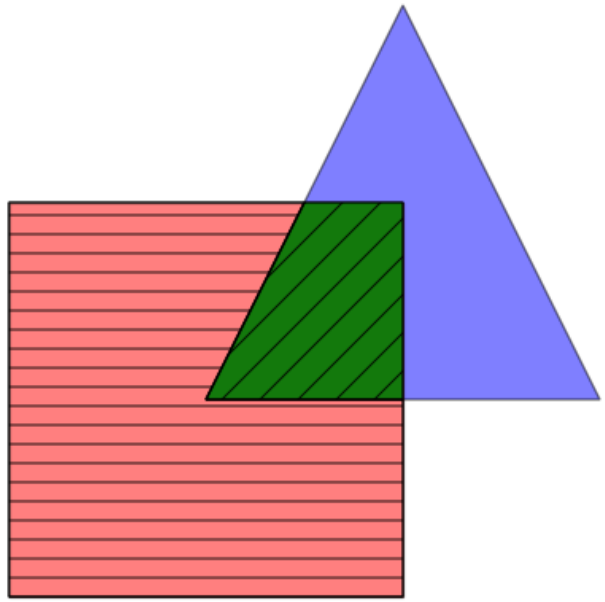
Vector data: Fiona and Shapely

```
# Initialize empty lists for later storage
cxList = []
cyList = []
areaList = []

for state in states: # loop over all features in collection
        # make temporary shapely geometry
        tmpShp = geometry.shape(state['geometry'])
        tmpX,tmpY = tmpShp.centroid.xy # x,y coords of centroid
        tmpArea = tmpShp.area # get geom's area
         # add data to lists for storage
        cxList.append(tmpX)
        cyList.append(tmpY)
        areaList.append(tmpArea)

plt.scatter(cxList,cyList,s=areaList*10)
plt.axis('tight')
plt.xlabel('Longitude',fontsize=16)
plt.ylabel('Latitude',fontsize=16)
plt.savefig('statesByArea.png',r=300)
plt.show()
plt.close()
```


60
50
40
30
20
Latitude
−160
−140
−120
−100
−80
Longitude

Vector data: Shapely operations

```
# Dummy data with simple geometries
square = geometry.Polygon([(0,0),(1,0),(1,1),(0,1)])
triangle = geometry.Polygon([(0.5,0.5),(1.5,0.5),(1,1.5)])
square.intesects(triangle) # Returns true if overlap
isect = square.intersection(triangle) # 'overlap'
diff = square.difference(triangle) # 'subtracts'
uni = square.union(triangle) # 'combines'
```



Vector data: Writing out

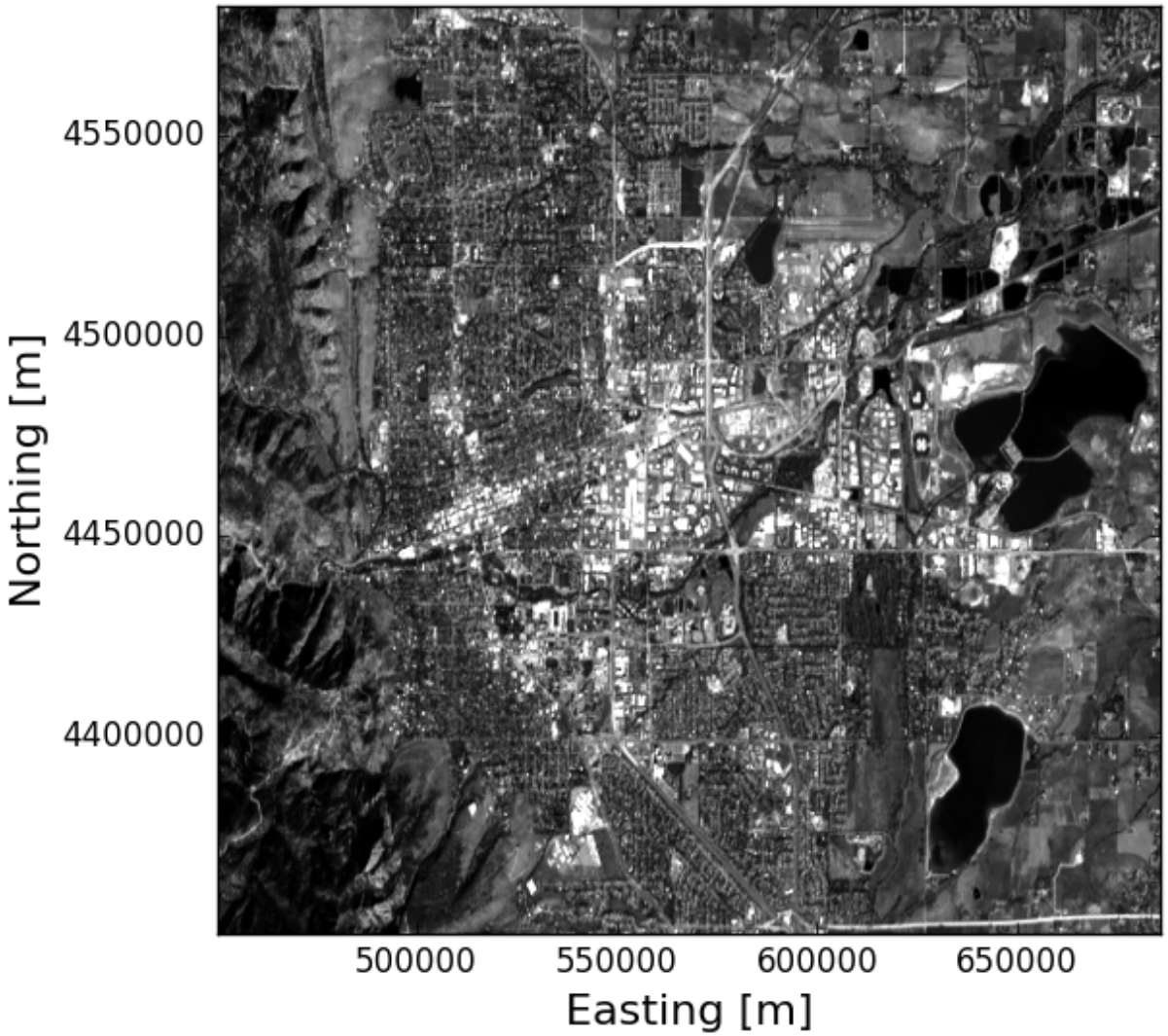
Writing out vector data.

Raster data: Plotting

Rasterio is a wrapper for GDAL, which, like OGR, is a little clunky and unintuitive to the average Python programmer. Opening, subsetting, and plotting an image with Rasterio and Matplotlib looks like:

```
src = rasterio.open(rasterFn) # open raster
# turn band 1 to numpy array; only use subset of rows and columns
sub = src.read(1,window=w) # w = ( (rMin,rMax), (cMin,cMax) )
# plot image; clim = min/max brightness; use 'gray' colormap
plt.imshow(sub,cmap='gray',clim=((bLow,bHigh)),
           extent=((xmin,xmax,ymin,ymax))) # x,y coords not row,col
plt.show()
```

Raster data: Plotting



Raster data: Transformation

Transforming between image coordinates and pixel coordinates is very easy.

```
row,col = src.index(x,y) # get pixel coords from ground coords
x,y = src.affine * (row,col) # get ground from pixel coord
```

As is finding out important image info.

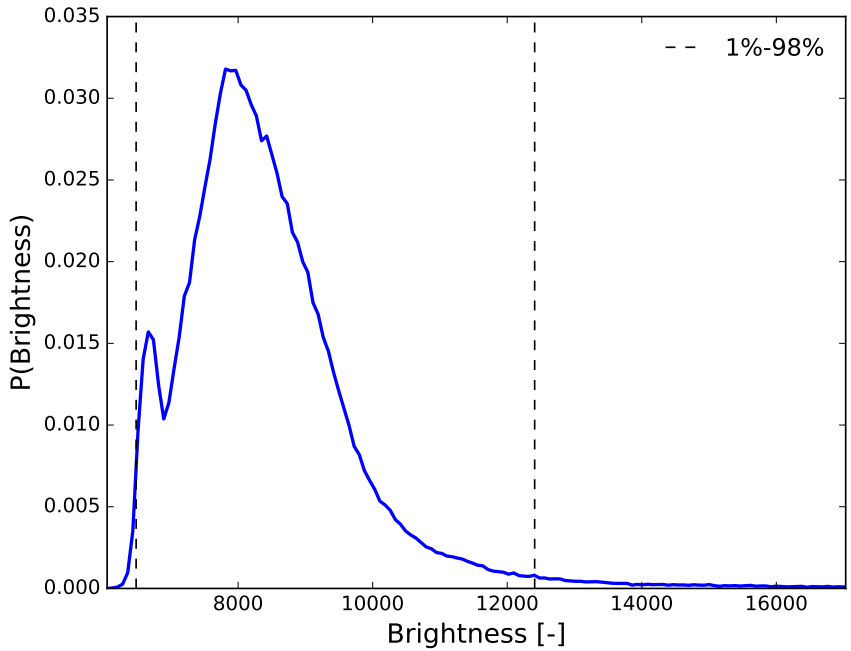
```
crs = src.crs # coordinate reference system
bnd = src.bounds # image bounding box coordinates
nRow = src.height # number of pixels N-S
nCol = src.height # number of pixels E-W
```

Raster data: Analysis and manipulation

You can do anything with an image-derived array as you would with a regular numpy array. For example,

- Measure value at a certain pixel (lat/lon) over time
- Histogram to get image characteristics
- Difference to images to get change over time
- Perform mathematical operations combining multiple arrays to create a new raster
- Image enhancement and processing using e.g., `scikit.image`, `cv2`, or `scipy`

Raster data: Analysis and manipulation




```
1 p2,p98 = np.percentile(sub,[2,98])
2 hist,edges = np.histogram(sub,bins=500)
```

Raster data: Writing

Writing out GeoTIFFs is straight forward. You create a new file with geospatial attributes (`crs`, `height`, etc.) and then specify with which dataset to populate the empty file.

```
'''
'w' = write, dtype = datatype (i.e., int, float)
crs = coordinate reference, driver = file format
transform = geographic transformation coefficients
'''
out = rasterio.open('boulderSub.tif', 'w', driver='GTiff',
                    height=sub.shape[0], width=sub.shape[1],
                    count=1, dtype=sub.dtype,
                    crs=src.crs, transform=src.transform)
out.write(sub,1) # write data to file
out.close() # 'flush' file
```

The `crs` and `transform` arguments can be taken from an existing dataset, as done here, or created anew.

`src.crs = CRS({'init':u'epsg:32613'})`

`src.transform = [450292.5,15.0,0.0,4582207.5,0.0,-15.0]`

Extensions: other useful packages

- **GDAL**: library underlying Rasterio. Has some really nice tools for working with DEMs.
- **OGR**: library underlying Shapely and Fiona. Has nice tools for translating vector data between formats.
- **PyProj**: projection and coordinate transformation.
- **GeoJSON**: storing geospatial information in a `dict`-like structure.
- **GeoPandas**: data structure and data analysis tool for geospatial data.
- **Descartes**: makes plotting Shapely geometries a little easier.

Thanks! william.armstrong@colorado.edu