Chapter No. 7
"Python and Elevation Data"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO.7 "Python and Elevation Data"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Joel Lawhead is a PMI-certified Project Management Professional (PMP) and the Chief Information Officer (CIO) for NVisionSolutions.com, an award-winning firm specializing in geospatial technology integration and sensor engineering.

He began using Python in 1997 and began combining it with geospatial software development in 2000. He has been published in two editions of the Python Cookbook by O'Reilly. He is also the developer of the widely used open source Python Shapefile Library (PyShp) and maintains the geospatial technical blog GeospatialPython.com and Twitter feed @SpatialPython discussing the use of the Python programming language within the geospatial industry.

In 2011, he reverse engineered and published the undocumented shapefile spatial indexing format and assisted fellow geospatial Python developer, Marc Pfister, in reversing the algorithm used, allowing developers around the world to create better-integrated and more robust geospatial applications involving shapefiles.

For More Information:
He has served as the lead architect, project manager, and co-developer for geospatial applications used by US government agencies including NASA, FEMA, NOAA, the US Navy, as well as many commercial and non-profit organizations. In 2002, he received the international "Esri Special Achievement in GIS" award for work on the Real-time Emergency Action Coordination Tool (REACT) for emergency management using geospatial analysis.

I would like to acknowledge my loving family including my wife Julie and four children Lauren, Will, Lillie, and Lainie who allowed me to write this book after hours. Thank you to my parents who inspired me through their actions to pursue computers, teaching, and writing; all the ingredients needed for a technical book. I would also like to acknowledge the work of the geospatial Python pioneers whose relentless and selfless contributions over the years in developing and publishing code to the geospatial Python body of knowledge made the content of this book possible, including Sean Gillies, Howard Butler, Matthew Perry, Frank Warmerdam, and Marc Pfister.

For More Information:
Learning Geospatial Analysis with Python

The best books change the way you look at the world. They take your mind to a different place than where you started. The transformation we experience from a good book is the reason books have survived for centuries as a way to share the breadth of human experience.

This book is about geospatial analysis. Geospatial analysis is the combination of statistical analysis, computational geometry, and image processing applied to data which is tied to the Earth (or even other planets). But that technical definition falls short of what geospatial analysis truly is. Similar to a good book, geospatial analysis tells a story about our world. This story is told through thematic maps, processed satellite images, and tables of information.

These stories quite literally change your worldview by revealing patterns about human behavior and natural processes that are otherwise difficult to discern or are even invisible to us. The increased awareness of our world and our place in it allows us to make better decisions about everything from agriculture to politics to disaster management.

This book will teach you geospatial analysis using the Python programming language. Python is a very popular and easy to learn language used in nearly every field. Python was invented in the late 1980s by Guido van Rossum and is based on the language "ABC" designed to teach programming to kids. The clean and intuitive syntax allows you to think about the problem you are trying to solve and not the language you are using. It also interfaces well with nearly every geospatial library available.

Learning Geospatial Analysis with Python supplements the library of Packt Publishing with a third book on geospatial technology and Python. The series offered by Packt Publishing covers the most complete range of published knowledge in this domain. In order to understand the scope of this book and its benefits, it helps to be familiar with the other offerings by Packt Publishing.

Python Geospatial Development by Erik Westra covers building desktop and web applications using Python and leading open source geospatial libraries. The focus of the book is capturing well-defined geospatial processes as requirements and then developing applications allowing users to interactively execute that process again and again.

Programming ArcGIS 10.1 with Python Cookbook by Eric Pimpler teaches readers how to automate ArcGIS 10.1, the leading Geographic Information System (GIS) software package by Esri. ArcGIS contains a Python environment called ArcPy that provides an interface to nearly the entire package. The book shows how to use Python to script the ArcGIS for a variety of geoprocessing tasks.

Geospatial analysis will allow you to look at the world in a whole new way and with new understanding. And Python will facilitate the journey and even make it fun! This book will serve as both a guide and future reference as you move deeper into this exciting field.

What This Book Covers

Chapter 1, Learning Geospatial Analysis with Python, introduces geospatial analysis as a way of answering questions about our world. The differences between GIS and remote sensing are explained. Common geospatial analysis processes are illustrated and a code for a simple geographic information system in Python is introduced.

Chapter 2, Geospatial Data, discusses geospatial data, and explains the forms geospatial data comes in. The most challenging part of geospatial analysis is acquiring the data you need and preparing it for analysis. This chapter explains the two major categories of data as well as several newer formats that are becoming more and more common. Familiarity with these data types is essential to understand geospatial analysis.

Chapter 3, The Geospatial Technology Landscape, covers the geospatial technology ecosystem that consists of thousands of software libraries and packages. This vast array of choices is overwhelming for newcomers to geospatial analysis. The secret to learning geospatial analysis quickly is to understand the handful of libraries and packages that really matter. Most other software is derived from these critical packages. Understanding the hierarchy of geospatial software and how it's used allows you to quickly comprehend and evaluate any geospatial tool.

Chapter 4, Geospatial Python Toolbox, explains the software and libraries introduced which forms the basis of the book and are used throughout. In this chapter, Python's role within the geospatial industry is elaborated: GIS scripting language, mash-up glue language, and full-blown programming language. Code examples are used to teach data editing concepts, and many of the basic geospatial concepts in Chapter 1, Learning Geospatial Analysis with Python, are also demonstrated in Python.

Chapter 5, Python and Geographic Information Systems, teaches the simple yet practical python GIS geospatial products using processes which can be applied to a variety of problems.

Chapter 6, Python and Remote Sensing, shows readers how to work with remote sensing geospatial data. Remote sensing includes some of the most complex and least
documented geospatial operations. This chapter will build a solid core for the reader and demystify remote sensing using Python.

Chapter 7, *Python and Elevation Data*, demonstrates the most common uses of elevation data, which can be contained in almost any geospatial format but is used quite differently from other types of geospatial data, and will show you how to work with its unique properties.

Chapter 8, *Advanced Geospatial Python Modeling*, discusses how geospatial data editing and processing help us understand the world as it is. But the true power of geospatial analysis is modeling. Geospatial models help us predict the future, narrow vast fields of choices down to the best options, and visualize concepts which cannot be directly observed in the natural world. This chapter uses Python to teach the reader the true power of geospatial technology.

Chapter 9, *Real-Time Data*, introduces real-time data and examines a modern phenomenon. A wise geospatial analyst once said, "As soon as a map is created it is obsolete." Until recently, by the time you collected data about the earth, processed it, and created a geospatial product, the world it represented had already changed. But modern geospatial data shatters this notion. Data sets are available over the Internet which are up to the minute or even the second. These data sets fundamentally change the way we perform geospatial analysis.

Chapter 10, *Putting It All Together*, combines the skills from previous chapters step-by-step to build a simple, automated geospatial analysis system which produces a report.

For More Information:  
Python and Elevation Data

Elevation data is one of the most fascinating types of geospatial data. It represents many different types of data sources and formats. Elevation data can display properties of both vector and raster data resulting in unique data products. Elevation data can serve the following purposes:

- Terrain visualization
- Land cover classification
- Hydrology modelling
- Transportation routing
- Feature Extraction

You can't perform all of these options with both raster and vector data but because elevation data is three dimensional, containing x, y, and z coordinates, you can often get more out of these data than any other type.

In this chapter, we're going to learn to read and write elevation data in both raster and vector point formats. We'll also create some derivative products. The topics we'll cover are:

- ASCII Grid elevation data files
- Shaded-relief images
- Elevation contours
- Gridding LIDAR data
- Creating a 3D mesh

For More Information:
ASCII Grid files

For most of this chapter we'll use ASCII Grid files or ASCIIGRID. These files are a type of raster data usually associated with elevation data. This grid format stores data as text in equally sized square rows and columns with a simple header. Each cell in a row/column stores a single numeric value, which can represent some feature of terrain, such as elevation, slope, or flow direction. The simplicity makes it an easy-to-use, platform independent raster format. This format is described in the ASCII GRIDS section in Chapter 2, Geospatial Data.

Throughout the book we've relied on GDAL and to some extent PIL to read and write geospatial raster data including the gdalnumeric module to load raster data into NumPy arrays. But ASCIIGRID allows us to read and write rasters using only Python or even NumPy.

As a reminder, some elevation data sets use image formats to store elevation data. Most image formats only support 8-bit values ranging between 0-255; however, some formats, including TIFF, can store larger values. Geospatial software can typically display these data sets; however, traditional image software and libraries usually do not. For simplicity in this chapter, we'll stick to the ASCIIGRID format for data, which is both human and machine readable, as well as being widely supported.

Reading grids

NumPy has the ability to read the ASCIIGRID format directly using its loadtxt() method designed to read arrays from text files. The first six lines consist of the header, which are not part of the array. The following lines are a sample of a grid header:

```
ncols        250
nrows        250
xllcorner    277750.0
yllcorner    6122250.0
cellsize     1.0
NODATA_value      -9999
```

For More Information:

Line 1 contains the number of columns in the grid, which is synonymous with the x axis. Line 2 represents the y axis described as a number of rows. Line 3 represents the x coordinate of the lower left corner, which is the minimum x value. Line 4 is the corresponding minimum y value in the lower left corner of the grid. Line 5 is the cell size or resolution of the raster. Because the cells are square, only one size value is needed, as opposed to the separate x and y resolution values in most geospatial rasters. The fifth line is no data value, which is a number assigned to any cell for which a value is not provided. Geospatial software ignores these cells for calculations and often allows special display settings for it, such as coloring them black. The value -9999 is a common no data placeholder value used in the industry, which is easy to detect in software. In some examples, we'll use the number zero; however, zero can often also be a valid data value.

The `numpy.loadtxt()` method includes an argument called `skiprows`, which allows you to specify a number of lines in the file to be skipped before reading array values. To try this technique out you can download a sample grid file called `myGrid.asc` at the following URL:

https://geospatialpython.googlecode.com/files/myGrid.asc

So for `myGrid.asc` we would use the following code:

```python
myArray = numpy.loadtxt("myGrid.asc", skiprows=6)
```

This line results in the variable `myArray` containing a `numpy` array derived from the ASCIIGRID file `myGrid.asc`. The ASC file name extension is used by the ASCIIGRID format. This code works great but there's one problem. NumPy allows us to skip the header but not keep it. And we need to keep it to have a spatial reference for our data for processing, as well as for saving this grid or creating a new one.

To solve this problem we'll use Python's built-in `linecache` module to grab the header. We could open the file, loop through the lines, store each one in a variable, and then close the file. But `linecache` reduces the solution to a single line. The following line reads the first line in the file into a variable called `line1`:

```python
import linecache
line1 = linecache.getline("myGrid.asc", 1)
```

In the examples in this chapter we'll use this technique to create a simple header processor that can parse these headers into python variables in just a few lines.
Writing grids

Writing grids in Numpy is just as easy as reading them. We use the corresponding `numpy.savetxt()` function to save a grid to a text file. The only catch is, we must build and add the six lines of header information before we dump the array to the file. This process is slightly different depending on if you are using NumPy versions before 1.7 or after. In either case, you build the header as a string first. If you are using NumPy 1.7 or later, the `savetxt()` method has an optional argument called header, which lets you specify a string as an argument. You can quickly check your NumPy version from the command line using the following command:

```
python -c "import numpy;print numpy.__version__"
1.6.1
```

The backwards compatible method is to open a file, write the header then dump the array. Here is a sample of the Version 1.7 approach to save an array called `myArray` to an ASCIIGRID file called `myGrid.asc`:

```python
header = "ncols        %s
" % myArray.shape[1]
header += "nrows        %s
" % myArray.shape[0]
header += "xllcorner    277750.0
"
header += "yllcorner    6122250.0
"
header += "cellsize     1.0
"
header += "NODATA_value      -9999
"
numpy.savetxt("myGrid.asc", myArray, header=header, fmt="%1.2f")
```

We make use of python format strings, which allow you to put placeholders in a string to format python objects to be inserted. The `%s` format variable turns whatever object you reference into a string. In this case we are referencing the number of columns and rows in the array. In NumPy, an array has both a size and shape property. The `size` property returns an integer for the number of values in the array. The `shape` property returns a tuple with the number of rows and columns, respectively. So, in the preceding example, we use the `shape` property tuple to add the row and column counts to the header of our ASCII Grid. Notice we also add a trailing newline character for each line (`\n`). There is no reason to change the x and y values, cell size, or nodata value unless we altered them in the script. The `savetxt()` method also has an `fmt` argument, which allows you to use Python format strings to specify how the array values are written. In this case the `%1.2f` value specifies floats with at least one number and no more than two decimal places.

For More Information:
The backwards compatible version for NumPy, before 1.6, builds the header string in the same way but creates the file handle first:

```python
import numpy
f = open("myGrid.asc", "w")
    f.write(header)
    numpy.savetxt(f, myArray, fmt="%1.2f")
    f.close()
```

In the examples in this chapter, we'll introduce Python with an approach for writing files, which provides more graceful file management by ensuring files are closed properly. If any exceptions are thrown, the file is still closed cleanly:

```python
with open("myGrid.asc", "w") as f:
    f.write(header)
    numpy.savetxt(f, myArray, fmt="%1.2f")
```

As you'll see in the upcoming examples, this ability to produce valid geospatial data files using only NumPy is quite powerful. In the next couple of examples we'll be using an ASCII grid Digital Elevation Model (DEM) of a mountainous area near Vancouver, British Columbia in Canada. You can download this sample as a ZIP file at the following URL:

https://geospatialpython.googlecode.com/files/dem.zip

The following image is the raw DEM colorized using QGIS with a color ramp that makes lower elevation values dark blue and higher elevation values bright red:
While we can conceptually understand the data this way, it is not an intuitive way to visualize the data. Let's see if we can do better.

**Creating a shaded relief**

Shaded relief maps color elevation in a way that it looks as if the terrain is cast in a low-angle light, which creates bright spots and shadows. The aesthetic styling creates an almost photographic illusion, which is easy to grasp to understand the variation in terrain. It is important to note that this style is truly an illusion as the light is often physically inaccurate and the elevation is usually exaggerated to increase contrast.

In this example, we'll use the ASCII DEM referenced previously to create another grid, which represents a shaded relief version of the terrain in NumPy. This terrain is quite dynamic so we won't need to exaggerate the elevation; however, the script has a variable called z, which can be increased from 1.0 to scale the elevation up.

After we define all the variables including input and output file names, you'll see the header parser based on the `linecache` module, which also uses a python list comprehension to loop and parse the lines that are then split from a list into the six variables. We also create a y cell size called `ycell`, which is just the inverse of the cell size. If we don't do this the resulting grid will be transposed.

Note we define file names for slope and aspect grids, which are two intermediate products that are combined to create the final product. These intermediate grids are output as well, just to take a look. They can also serve as inputs to other types of products.

This script uses a three by three windowing method to scan the image and smooth out the center value in these mini grids. But because we are using NumPy, we can process the entire array at once, as opposed to a lengthy series of nested loops. This technique is based on the excellent work of a developer called Michal Migurski, who implemented the clever NumPy version of Matthew Perry's C++ implementation, which served as the basis for the DEM tools in the GDAL suite.

After the slope and aspect are calculated, they are used to output the shaded relief. Finally, everything is saved to disk from NumPy. In the `savetxt()` method we specify a 4 integer format string, as the peak elevations are several thousand feet:

```python
from linecache import getline
import numpy as np

# File name of ASCII digital elevation model
source = 'dem.asc'
# File name of the slope grid
```
slopegrid = "slope.asc"
# File name of the aspect grid
aspectgrid = "aspect.asc"
# Output file name for shaded relief
shadegrid = "relief.asc"
## Shaded elevation parameters
# Sun direction
azimuth=315.0
# Sun angle
altitude=45.0
# Elevation exageration
z=1.0
# Resolution
scale=1.0
# No data value for output
NODATA = -9999

# Needed for numpy conversions
deg2rad = 3.141592653589793 / 180.0
rad2deg = 180.0 / 3.141592653589793

# Parse the header using a loop and
# the built-in linecache module
hdr = [getline(source, i) for i in range(1,7)]
values = [float(h.split(" ")[0][-1].strip()) for h in hdr]
cols,rows,lx,ly,cell,nd = values
xres = cell
yres = cell * -1

# Load the dem into a numpy array
arr = np.loadtxt(source, skiprows=6)

# Exclude 2 pixels around the edges which are usually NODATA.
# Also set up structure for a 3x3 window to process the slope
# throughout the grid
window = []
for row in range(3):
    for col in range(3):

# Process each cell
```
window[3] + z * window[6]) - \
(z * window[2] + z * window[5] + z * \nwindow[5] + z * window[8])) / (8.0 * xres * scale);


# Calculate slope
slope = 90.0 - np.arctan(np.sqrt(x*x + y*y)) * rad2deg

# Calculate aspect
aspect = np.arctan2(x, y)

# Calculate the shaded relief
shaded = np.sin(altitude * deg2rad) * np.sin(slope * deg2rad) \n+ np.cos(altitude * deg2rad) * np.cos(slope * deg2rad) \n* np.cos((azimuth - 90.0) * deg2rad - aspect);
shaded = shaded * 255

# Rebuild the new header
header = "ncols %s
nrows %s
xllcorner %s
yllcorner %s
cellsize %s
NODATA_value %s"

# Set no-data values
for pane in window:
slope[pane == nd] = NODATA
aspect[pane == nd] = NODATA
shaded[pane == nd] = NODATA

# Open the output file, add the header, save the slope grid
with open(slopegrid, "wb") as f:
f.write(header)
np.savetxt(f, slope, fmt="%4i")
```

For More Information:
# Open the output file, add the header, save the slope grid
with open(aspectgrid, "wb") as f:
    f.write(header)
    np.savetxt(f, aspect, fmt="%4i")

# Open the output file, add the header, save the array
with open(shadegrid, "wb") as f:
    f.write(header)
    np.savetxt(f, shaded, fmt="%4i")

If we load the output grid into QGIS and specify the styling to stretch the image to the min and max, we see the following image. You can also open the image in the FWTools OpenEV application discussed in the *Installing GDAL* section in *Chapter 4, Geospatial Python Toolbox*, which will automatically stretch the image for optimal viewing.

For More Information:
As you can see, the preceding image is much easier to comprehend than the original pseudo-color representation we examined originally. Next, let's look at the slope raster used to create the shaded relief:

The slope shows the gradual decline in elevation from high points to low points in all directions of the data set. Slope is an especially useful input for many types of hydrology models.
The aspect shows the maximum rate of downslope change from one cell to its neighbors. If you compare the aspect image to the shaded relief image you can see the red and gray values of the aspect image correspond to shadows in the shaded relief. So the slope is primarily responsible for turning the DEM into a terrain relief while the aspect is responsible for the shading.

Creating elevation contours

Now let's look at another way to better visualize elevation using contours. A contour is an isoline along the same elevation in a data set. Contours are usually stepped at intervals to create an intuitive way to represent elevation data, both visually and numerically, using a resource efficient vector data set.

The input for generating contours is our DEM and the output is a shapefile. The algorithm for generating contours is fairly complex and very difficult to implement using NumPy's linear algebra. So our solution in this case is to fall back on the GDAL library, which has a contouring method available through the Python API. In fact, the majority of this script is just setting up the OGR library code needed to output shapefile. The actual contouring is a single method call named `gdal.ConTourGenerate()`. Just before that call, there are comments defining the method's arguments. The most important ones are as follows:

- `contourInterval`: It is the distance in data set units between contours
- `contourBase`: It is the starting elevation for contouring
- `fixedLevelCount`: It specifies a fixed number of contours as opposed to distance
- `idField`: It is a name for a required shapefile dbf field, usually just called ID
- `elevField`: It is a name for a required shapefile dbf field for the elevation value useful for labeling in maps

You should have GDAL and OGR installed from *Installing GDAL* section in Chapter 4, *Geospatial Python Toolbox*. In the following code we will define the input DEM file name, the output shapefile name, create the shapefile data source with OGR, get the OGR layer, open the DEM, and generate contours on the OGR layer:

```python
import gdal
import ogr

# Elevation DEM
source = "dem.asc"
# Output shapefile
target = "contour"
```

For More Information:
Python and Elevation Data

ogr_ds = ogr.GetDriverByName('ESRI Shapefile').CreateDataSource(target + ".shp")
ogr_lyr = ogr_ds.CreateLayer(target, geom_type = ogr.wkbLineString25D)
field_defn = ogr.FieldDefn('ID', ogr.OFTInteger)
ogr_lyr.CreateField(field_defn)
field_defn = ogr.FieldDefn('ELEV', ogr.OFTReal)
ogr_lyr.CreateField(field_defn)

# gdal.ContourGenerate() arguments
# Band srcBand,
# double contourInterval,
# double contourBase,
# double[] fixedLevelCount,
# int useNoData,
# double noDataValue,
# Layer dstLayer,
# int idField,
# int elevField

ds = gdal.Open('dem.asc')
gdal.ContourGenerate(ds.GetRasterBand(1), 400, 10, [], 0, 0, ogr_lyr, 0, 1)

Now let's draw the contour shapefile we just created using PNGCanvas, introduced in the PNGCanvas section of Chapter 4, Geospatial Python Toolbox.

import shapefile
import pngcanvas

# Open the contours
r = shapefile.Reader("contour.shp")
# Setup the world to pixels conversion
xdist = r.bbox[2] - r.bbox[0]
ydist = r.bbox[3] - r.bbox[1]
iwidth = 800
iheight = 600
xratio = iwidth/xdist
yratio = iheight/ydist
contours = []
# Loop through all shapes
for shape in r.shapes():
    # Loop through all parts
    for i in range(len(shape.parts)):
        pixels=[]
pt = None
if i < len(shape.parts) - 1:
    pt = shape.points[shape.parts[i]:shape.parts[i + 1]]
else:
    pt = shape.points[shape.parts[i]:]
for x, y in pt:
    px = int(iwidth - ((r.bbox[2] - x) * xratio))
    py = int((r.bbox[3] - y) * yratio)
    pixels.append([px, py])
contours.append(pixels)
# Set up the output canvas
canvas = pngcanvas.PNGCanvas(iwidth, iheight)
# PNGCanvas accepts rgba byte arrays for colors
red = [0xff, 0, 0, 0xff]
canvas.color = red
# Loop through the polygons and draw them
for c in contours:
    canvas.polyline(c)
# Save the image
f = open("contours.png", "wb")
f.write(canvas.dump())
f.close()

We end up with the following image:
If we bring our shaded relief ASCIIGRID and the shapefile into a GIS, such as QGIS, we can create a simple topographic map as follows. You can use the elevation dbf field you specified in the script to label the contour lines with the elevation.

The techniques in these NumPy grid examples provide the building blocks for all kinds of elevation products. The USGS has an excellent web page, with sample elevation-based data layers, including the examples we created as well as some more advanced types:


Next we’ll work with one of the most complex elevation data types: LIDAR data.

**Working with LIDAR**

LIDAR stands for Light Detection and Ranging. It is similar to radar-based images but uses finite laser beams, which hit the ground hundreds of thousands of times per second to collect a huge amount of very fine (x, y, z) locations as well as time and intensity. The intensity value is what really separates LIDAR from other data types. For example, but the asphalt roof top of a building may be the same elevation as the top of a nearby tree, the intensities will be different. And just like remote sensing radiance values in a multispectral satellite image allow us to build classification libraries, the intensity values of LIDAR data allow us to classify and colorize LIDAR data as well.

For More Information:
The high volume and precision of LIDAR actually make it difficult to use. A LIDAR data set is referred to as a point cloud because the shape of the data set is usually irregular, as the data is three dimensional with outlying points. There are not many software packages which effectively visualize point clouds. Furthermore, an irregular shaped collection of finite points is just hard to interact with, even when using appropriate software.

For these reasons, one of the most common operations on LIDAR data is to project the data and resample it to a regular grid. We'll do exactly that using a small LIDAR data set. This data set is approximately 7 mb uncompressed, and contains over 600,000 points. The data captures some easily identifiable features, such as buildings, trees, and cars in parking lots. You can download the zipped data set at the following URL:

https://geospatialpython.googlecode.com/files/lidar.zip

The file format is a very common binary format specific to LIDAR called LAS. Unzip this file to your working directory. In order to read this format, we'll use a pure Python library called Laspy. You can install it from PyPI:

```
easy_install laspy
```

Or

```
pip install laspy
```

### Creating a grid from LIDAR

With laspy installed, we are ready to create a grid from LIDAR. This script is fairly straightforward. We loop through the (x,y) point locations in the LIDAR data and project them onto our grid with a cell size of 1 meter. Because of the precision of the LIDAR data, we'll end up with multiple points in a single cell. We average these points to create a common elevation value. Another issue we have to deal with is data loss. Any time you resample data, you lose information. In this case we'll end up with no data holes in the middle of the raster. To deal with this issue, we fill these holes with average values from surrounding cells, which is a form of interpolation.

We only need two modules, both available on PyPI, as shown in the following code:

```python
from laspy.file import File
import numpy as np

# Source LAS file
source = 'lidar.las'

# Output ASCII DEM file
```
target = "lidar.asc"

# Grid cell size (data units)
cell = 1.0

# No data value for output DEM
NODATA = 0

# Open LiDAR LAS file
las = File(source, mode="r")

# xyz min and max
min = las.header.min
max = las.header.max

# Get the x axis distance
xdist = max[0] - min[0]

# Get the y axis distance
ydist = max[1] - min[1]

# Number of columns for our grid
cols = int(xdist) / cell

# Number of rows for our grid
rows = int(ydist) / cell

cols += 1
rows += 1

# Track how many elevation values we aggregate
count = np.zeros((rows, cols)).astype(np.float32)

# Aggregate elevation values
zsum = np.zeros((rows, cols)).astype(np.float32)

# Y resolution is negative
ycell = -1 * cell

# Project x, y values to grid
projx = (las.x - min[0]) / cell
projy = (las.y - min[1]) / ycell

# Cast to integers and clip for use as index
ix = projx.astype(np.int32)
iy = projy.astype(np.int32)

# Loop through x, y, z arrays, add to grid shape, # and aggregate values for averaging
for x, y, z in np.nditer([ix, iy, las.z]):
    count[y, x] += 1
    zsum[y, x] += z

# Change 0 values to 1 to avoid numpy warnings, # and NaN values in array
nonzero = np.where(count > 0, count, 1)
# Average our z values
zavg = zsum / nonzero

# Interpolate 0 values in array to avoid any holes in the grid
mean = np.ones((rows, cols)) * np.mean(zavg)
left = np.roll(zavg, -1, 1)
lavg = np.where(left > 0, left, mean)
right = np.roll(zavg, 1, 1)
ravg = np.where(right > 0, right, mean)
interpolate = (lavg + ravg) / 2
fill = np.where(zavg > 0, zavg, interpolate)

# Create our ASCII DEM header
header = "ncols        %s
nrows        %s
xllcorner    %s
yllcorner    %s
cellsize     %s
NODATA_value      %s"
header += fill.shape[1]
header += fill.shape[0]
header += min[0]
header += min[1]
header += cell
header += NODATA

# Open the output file, add the header, save the array with open(target, "wb") as f:
    f.write(header)
    # The fmt string ensures we output floats
    # that have at least one number but only # two decimal places
    np.savetxt(f, fill, fmt="%1.2f")
The result of our script is an ASCIIGRID, which looks like the following image when viewed in OpenEV. Higher elevations are lighter while lower elevations are darker. Even in this form you can see buildings, trees, and cars.

If we assigned a heat map color ramp, the colors give you a sharper sense of the elevation differences:
So what happens if we run this output DEM through our shaded relief script from earlier? There’s a big difference between straight-sided buildings and sloping mountains. If you change the input and output names in the shaded relief script to process the LIDAR DEM we get the following result:

![Image of shaded relief output]

The gently rolling slope of the mountainous terrain is reduced to outlines of major features in the image. And in the aspect image the changes are so sharp and over such short distances that the output image is very chaotic to view as shown in the following screenshot:

![Image of aspect output]
But despite the difference in these images and the coarser but smoother mountain versions, we still get a very nice shaded relief, which somewhat visually resembles a black and white photograph:

![Shaded Relief Image]

### Using PIL to visualize LIDAR

The previous DEM images in this chapter were visualized using QGIS and OpenEV. But we can also create output images in Python by introducing some new functions of the **Python Imaging Library (PIL)**, which we didn't use in previous chapters. In this example we'll use the `PIL.ImageOps` module, which has functions for histogram equalization and automatic contrast enhancement. We'll use PIL's `fromarray()` method to import the data from NumPy. Let's see how close we can get to the output of the desktop GIS programs pictured in this chapter with the help of the following code:

```python
import numpy as np
import Image
import ImageOps

# Source LAS file
source = 'relief.asc'

# Output ASCII DEM file
target = 'relief.bmp'
```

For More Information:
# Load the ASCII DEM into a numpy array
arr = np.loadtxt(source, skiprows=6)

# Convert array to numpy image
im = Image.fromarray(arr).convert('RGB')

# Enhance the image:
# equalize and increase contrast
im = ImageOps.equalize(im)
im = ImageOps.autocontrast(im)

# Save the image
im.save(target)

As you can see in the following screenshot, the enhanced shaded relief has a sharper
relief than the previous version:
Now let’s colorize our shaded relief. We’ll use the built-in Python `colorsys` module for color space conversion. Normally, we specify colors as RGB values. But to create a color ramp for a heat map scheme we’ll use HSV values, which stand for **Hue, Saturation, Value** to generate our colors. The advantage of HSV is you can tweak the **H** value as a degree between zero and 360 on a color wheel. Using a single value for hue allows you to use a linear ramping equation, which is much easier than trying to deal with combinations of three separate RGB values. The following image from the online magazine *Qt Quarterly* illustrates the HSV color model:

![HSV Color Model](image)

The `colorsys` module lets you switch back and forth between HSV and RGB values. The module returns percentages for RGB values, which then must be mapped to the 0-255 scale for each color.

In the following code we’ll convert the ASCII DEM to a PIL image, build our color palette, apply the color palette to the grayscale image, and save the image:

```python
import numpy as np
import Image
import ImageOps
import colorsys

# Source LIDAR file
source = 'lidar.asc'

# Output image file
target = 'lidar.bmp'

# Load the ASCII DEM into a numpy array
```
arr = np.loadtxt(source, skiprows=6)

# Convert the numpy array to a PIL image
im = Image.fromarray(arr).convert('L')

# Enhance the image
im = ImageOps.equalize(im)
im = ImageOps.autocontrast(im)

# Begin building our color ramp
palette = []

# Hue, Saturation, Value
# color space
h = .67
s = 1
v = 1

# We'll step through colors from:
# blue-green-yellow-orange-red.
# Blue=low elevation, Red=high elevation
step = h/256.0

# Build the palette
for i in range(256):
    rp, gp, bp = colorsys.hsv_to_rgb(h, s, v)
    r = int(rp*255)
    g = int(gp*255)
    b = int(bp*255)
    palette.extend([r, g, b])
    h -= step

# Apply the palette to the image
im.putpalette(palette)

# Save the image
im.save(target)

For More Information:
The code produces the following image with higher elevations in warmer colors and lower elevations in cooler colors:

![Image with elevation data](image)

In this image we actually get more variation than the QGIS version. We could potentially improve this image with a smoothing algorithm. But as you can see, we have the full range of our color ramp expressed from cool to warm colors as the elevation change increases.

**Creating a Triangulated Irregular Network (TIN)**

The following example is our most sophisticated example yet. A Triangulated Irregular Network or TIN is a vector representation of a point data set in a vector surface of points connected as triangles. The most common type of TIN is based on Delaunay triangulation, which includes all points without redundant triangles. The purpose of the TIN is to use vector data that requires storing fewer points than an equivalent raster data set. It can also be generated on the fly for streaming applications in which you move around interactively in the data set, so the entire terrain isn't visible all at once.

For More Information:

The Delaunay triangulation is very complex. We'll use a pure Python library written by Bill Simons as a part of Steve Fortune's Delaunay triangulation algorithm called `voronoi.py` to calculate the triangles in our LIDAR data. You can download the script to your working directory from the following URL: https://geospatialpython.googlecode.com/files/voronoi.py

This script reads the LAS file, generates the triangles, then loops through them and writes out a shapefile. For this example, we'll use a clipped version of our LIDAR data to reduce the area for processing. If we run our entire data set of 600,000 plus points, the script will run for hours and generate over half a million triangles. You can download the clipped LIDAR data set as a zip file at the following URL:

https://geospatialpython.googlecode.com/files/clippedLAS.zip

We have several status messages, which print while the script runs. We also use the Python built-in cPickle module to save our triangles, and shapefile objects to speed up future runs. Unzip the LAS file and run the following code to generate a shapefile called `mesh.shp`:

```python
import cPickle
import os
import time
import math
# Third-party Python modules:
import numpy as np
import shapefile
from laspy.file import File
import voroni

# Source LAS file
source = "clippedLAS.las"

# Output shapefile
target = "mesh"

# Triangles archive
archive = "triangles.p"

# Pyshp archive
pyshp = "mesh_pyshp.p"

# Point class required by # the voroni module
class Point:
  def __init__(self, x, y):
    self.px = x
    self.py = y
```

For More Information:
def x(self):
    return self.px

def y(self):
    return self.py

# This will be the triangle
# array. Load it from a pickle
# file or use the voroni module
# to create the triangles.
triangles = None

if os.path.exists(archive):
    print "Loading triangle archive..."
    f = open(archive, "rb")
    triangles = cPickle.load(f)
    f.close()
    # Open LIDAR LAS file
    las = File(source, mode="r")
else:
    # Open LIDAR LAS file
    las = File(source, mode="r")
    points = []
    print "Assembling points..."
    # Pull points from LAS file
    for x, y in np.nditer((las.x, las.y)):
        points.append(Point(x, y))
    print "Composing triangles..."
    # Delaunay Triangulation
    triangles = voroni.computeDelaunayTriangulation(points)
    # Save the triangles to save time if we write more than
    # one shapefile.
    f = open(archive, "wb")
    cPickle.dump(triangles, f, protocol=2)
    f.close()

print "Creating shapefile..."
w = None
if os.path.exists(pyshp):
    f = open(pyshp, "rb")
    w = cPickle.load(f)
    f.close()
else:
    # PolygonZ shapefile (x, y, z, m)
    w = shapefile.Writer(shapefile.POLYGONZ)
    w.field("X1", "C", "40")
    w.field("X2", "C", "40")
    w.field("X3", "C", "40")
    w.field("Y1", "C", "40")
    w.field("Y2", "C", "40")
    print "For More Information:
w.field("Y3", "C", "40")
w.field("Z1", "C", "40")
w.field("Z2", "C", "40")
w.field("Z3", "C", "40")

tris = len(triangles)
# Loop through shapes and track progress every 10 percent
last_percent = 0
for i in range(tris):
    t = triangles[i]
    percent = int((i/(tris*1.0))*100.0)
    if percent % 10.0 == 0 and percent > last_percent:
        last_percent = percent
        print "%s %% done - Shape %s/%s at %s" % (percent, i, tris, time.ctime())
    part = []
    x1 = las.x[t[0]]
y1 = las.y[t[0]]
z1 = las.z[t[0]]
x2 = las.x[t[1]]
y2 = las.y[t[1]]
z2 = las.z[t[1]]
x3 = las.x[t[2]]
y3 = las.y[t[2]]
z3 = las.z[t[2]]
# Check segments for large triangles along the convex hull which is an common artificat in Delaunay triangulation
max = 3
if math.sqrt((x2-x1)**2+(y2-y1)**2) > max: continue
if math.sqrt((x3-x2)**2+(y3-y2)**2) > max: continue
if math.sqrt((x3-x1)**2+(y3-y1)**2) > max: continue
    part.append([x1,y1,z1,0])
    part.append([x2,y2,z2,0])
    part.append([x3,y3,z3,0])
    w.poly(parts=[part])
    w.record(x1,x2,x3,y1,y2,y3,z1,z2,z3)
print "Saving shapefile..."
# Pickle the Writer in case something goes wrong. Be sure to delete this file to recreate teh shapefile.
f = open(pyshp, "wb")
cPickle.dump(w, f, protocol=2)
f.close()
w.save(target)
print "Done."

For More Information:
The following image shows a zoomed in version of the TIN over the colorized LIDAR data:

Summary

Elevation data can often provide a complete data set for analysis and derivative products without any other data. In this chapter we learned to:

- Read/write ASCII Grids using only NumPy
- Create shaded reliefs, slope grids, and aspect grids
- Create elevation contours
- Transform LIDAR data into a grid
- Visualize LIDAR data with PIL
- Create a TIN

In the next chapter we'll combine the building blocks from the previous three chapters to do some advanced modeling and actually create some information products.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.