DEVELOPMENT AND VALIDATION OF A LIBRARY FOR ITERATIVE WINDOW-BASED PROCESSING OF GEOSPATIAL DATA

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

David Michael Schwartz

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

April 2021

Fargo, North Dakota

NORTH DAKOTA STATE UNIVERSITY

Graduate School

Title

DEVELOPMENT AND VALIDATION OF A LIBRARY FOR ITERATIVE WINDOW-BASED PROCESSING OF GEOSPATIAL DATA

By

David Michael Schwartz

The supervisory committee certifies that this thesis complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Anne Denton
Chair

Dr. Simone Ludwig
Co-Chair

Dr. Azer Akhmedov
Co-Chair

Approved:

April 15, 2021	Dr. Simone Ludwig
Date	Department Chair

ABSTRACT

High-resolution spectral images and digital elevation models are widely available. With this quantity of data, it is imperative to develop fast algorithms to extract information. We present a Python library that implements a set of algorithms for aggregating data within sliding windows. The algorithms have $O(\log(n))$ time complexity and maintain the original image resolution. They are vectorized and written with NumPy to create fast code with C-like performance. The library offers several analysis procedures, architected such that additional procedures utilizing sliding windows can easily be added. Slope, aspect, and curvature analyses exist for digital elevation models. Fractal dimensions and correlation analyses are also present to be used on a range of different images. The software architecture of the library is outlined and motivated. It includes visualized comparisons of analyses and unit testing. Testing procedures are implemented using analytical results from Wolfram Mathematica combined with brute-force algorithms.

ACKNOWLEDGEMENTS

Thank you to my wonderful advisor, Dr. Anne Denton. She took me under her wing and inspired me to push through the hardest parts of writing a thesis. Thank you to my committe members. Dr. Azer Akhmedov, who is also an excellent math advisor, and Dr. Simone Ludwig, who showed me the fascinating world of computer architecture.

DEDICATION

In memory of Jack Schwartz, my loving grandfather and biggest fan.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1. Geographic Information Systems

The simplest definition of geospatial data is any data that has a geographic location associated with it. These locations can be either global coordinates or addresses. According to Behm et. al. [2], geospatial data is used in almost every industry, "Geospatial data has become an integral element in how companies and organizations conduct business around the world.". The uses are as varied as choosing the right store placement in retail to finding the best route for emergency vehicles in the transportation industry.

In remote sensing a distinction is made between active and passive techniques. Sensing generally involves sensing the wave lengths of light bouncing off of the observed surfaces. This is called the *spectral signature* of the surfaces. This type of sensing results in spectral images, where data in an image is layered in *bands*. For the typical consumer camera, these are red, green, and blue bands. Active sensing involves providing an energy souce and sensing that same energy when it bounces back. Passive techniques provides no such energy source and simply senses the energy available in the environment. Another type of sensing involves active sensors measuring the time it takes their energy to bounce back. Such as for light detection and ranging (LiDAR) data, which we will be using in the form of digital elevation models (DEMs) in this thesis.

Remotely sensed data can have spatial resolutions in the range of centimeters to kilometers. The spatial resolution describes the linear dimension of each pixel. This resolution depends on the quality of the camera sensor and the altitude of the device. As stated by Allan [1], "The higher the resolution of the imagery, the more man made objects that can be identified. The human eye – the best image processor of all – can quickly detect and identify these objects.". Although, higher resolution isn't a complete boon when we consider that computers, and not the human eye, are tasked with identifying these objects, "If the same techniques that were developed for earlier lower resolution satellite imagery are used on the high-resolution imagery, (such as maximum likelihood classification), the results can actually create a negative impact.". The new information gives a much more fined grained representation of the surfaces, and must be accounted for. Objects will have more precise shapes and more discrete colors. With lower resulution images, objects can often be simplified to simple shapes and colors. For example, a road may be a straight line with a single color in a low resultion image, but in a higher resolution it will have many curves and discrete colors for the roadway, lines, and shoulder.

Geospatial data is a broad term that encompasses all data that has any geographic component. GIS stands for Geographic Information System. It traditionally refers to a software system built on top of a database of geospatial and attribute data. GIS enables visualizations and manipulation of the stored data. It can be used to process vector or raster geospatial data. Vector data is defined by points, lines, or other shapes. Raster data is defined by grids cells with discrete or continuous values. The attribute data is more akin to traditional databases, represented in columns and rows of a table. The attribute data is generally auxilliary information for the geospatial data. It includes information such as names or construction dates of different man made geographic features.

The main function of GIS is the combination and analysis of many types of geospatial data. Dempsey [4] says, "On the most basic level, geographic information systems technology is used as computer cartography, that is for straight forward map making. The real power of GIS, however, is through using spatial and statistical methods to analyze attribute and geographic information. The end result of the analysis can be derivative information, interpolated information or prioritized information.". Dempsey mentions that examples of such analyses include: calculating relative distances with buffer analysis and detecting types of vegetation with Normalized Difference Vegetation Index (NDVI). The NDVI analysis is actually included in this thesis.

Among the myriad ways of obtaining remote sensing data, satellite data is very common. The longest running satellite program is the Landsat program created by the United States Geologial Survey (USGS) and the National Aeronautics and Space Administration (NASA). The first Landsat was launched in 1972. NASA has continued to relase new missions since then, with a plan to release Landsat 9 in 2021 [17]. In 2009, a decade after the launch of Landsat 7 in 1999, these images became freely available to the public [13]. While the Landsat missions 1-3 touted 60m resolutions, all subsequent missions have had 30m spatial resolutions [18]. For satellites with higher spatial resolution, there exists the field of small satellites. These satellites can range from 1m to 10m in spatial resolution [11]. The highest resolution available is going to be that of Unmanned

Aerial vehicles (UAVs). These devices are much more flexible by their very nature of not being put into orbit. They can have very high resolutions of up to .1m [19].

There exists a need for more robust statistical methods in GIS. According to Nelson [14], there is an opinion among established researchers that the following exist as future opportunites for improvement in spacial statistics: "integration of GIS and spatial analysis", "methods for large and multitemporal data sets", "advancing local spatial statistics", "communicating spatial analysis results collectively", and "geography as the home for spatial analysis".

Remotely sensed DEMs can be used to calculate the slope, aspect, and curvature of a surface. According to two approaches to ranking hill slope algorithms, the best performing are those by Sharpnack & Akin's and Fleming & Hoffer's [10]. Both of these algorithms use small 3x3 pixel windows. The Sharpnack & Askin algorithm [15] subtracts the sum of the 3 pixels along one edge of the 3x3 window from the sum of the 3 pixels on the opposite edge. The resulting value is used for the slope in one direction. The opposing direction is calculated in the same fashion. The Fleming & Hoffer's algorithm [7] follows a very similar process with the exception that it only uses the four ortholinear pixels. These are common techniques used in GIS software for topographical calculations.

Typically these $3x3$ windows are used. Sometimes even larger windows of sizes $5x5$ or $7x7$ are used. Even with these larger sizes, it is still common to parse the entire window. We will refer to these types of algorithms as *brute force.* The computational time of these algorithms generally scales n^2 with the input size n. When resolutions are very high, the $3x3$ pixel area may not represent much area at all. When we desire to increase the scale of the pixels, images require resampling. This involves combining pixels, reducing the resolution of the image and making each pixel represent a larger area. This helps us achieve our desired computation, but defeats the purpose of our higher resolution image!

Other quantities that can be derived using window-based algorithms include fractal dimensions. Fractal dimensions were originally proposed by Mandelbrot in his curiously simple question of "How long is the coast of Britain?" [12]. A common method used to calcuate fractal dimensions is the box counting method, first proposed by Gagenpain et. al. [8]. This is where an image is broken up into grids of increasingly smaller size, and the number of boxes counted in each grid is plotted against the scale of the grid. The regression slope of such a plot would be the theoretical fractal dimension. Such a calculation lends itself very well to the algorithm discussed later in this thesis, where simple aggregations and regressions are all that is required.

1.2. Problem Statement

Algorithms have been proposed to help solve some of the issues with topographical calculations. A sliding-window based algorithm has been developed by Denton et. al. [5], originally for computing regression coefficients between bands. Several uses of the algorithm have been proposed by Denton, including slope, curvature, and fractal dimensions. Algorithms of this nature are able to do simple aggregations involving operations such as max, min, and sum in $O(\log(n))$ time complexity and don't reduce the resolution of the image. These algorithms have been developed in isolation and have not been combined into a single library or integrated into an existing GIS software suite such as ArcGIS. In this thesis, we propose a software library that combines the uses of the sliding window algorithm into a consistent package.

The algorithm works by reusing aggregates in each iteration. Four non-overlapping aggregates are added from the previous iteration, quadrupling the total number of aggregated pixels. Aggregates are stored in the top left pixel of a sliding window. In each iteration, the top left corners of four aggregation windows are added. These result in aggregates of windows 2^n pixels long, with n being the current aggregation number starting at one. We desire to aggregate four non-overlapping adjacent windows every iteration, to form new windows with quadruple the previous previous amount of aggregated pixels. Crucial parts of the algorithm are review in Section 2.1.

The library we developed for the sliding-window algorithm was written as an open source Python package. It utilized the NumPy Python package for high performance sliding-window computations. NumPy was actually written in C and uses Single Instruction, Multiple Data (SIMD) CPU architecture for vectorized array operations. Other packages include Rasterio for GIS raster image manipulation, Matplotlib for result visualization, Scikit-learn for clustered results, and Seaborn for pairplotted results.

The library is broken up into five basic parts: A high level class for general image maniplation tasks, an aggregation file for sliding window operations, two files for analysis formulae, a file for unit tests, and a file for combined visualizations of multiple different analyses. The high level class abstracts away the complexities of analyzing images. In the simplest case, all it requires is a file path and a function call to analyze an image. The aggregation file encapsulates the sliding window algorithms, so that they are able to be resused in a variety of different analyses. This file includes brute-force versions of the functions for testing purposes as well. The files containing the formulas are separated based on file type. One for DEMs called dem.py and one for spectral images called rbg.py. The DEM file requires the user to aggregate the arrays beforehand, but rbg.py does the aggregations for the user. The unit test file ensures the validity of our operations. Currently, all DEM and aggregation operations have been validated. Finally, the cluster file enables the combined visualization of analyses. It specifies the visualization by function call and by the different arguments passed in, such as aggregation numbers, analysis types, spectral bands, and so on. These parts are covered in Section 2.2.

Unit tests in this library are performed with brute-force operations and analytical results derived in Wolfram Mathematica. For DEM tests, the derived formulae of a Gaussian hill are compared against the sliding window results of an artificially generated Guassian hill image. Results from the library are iterated though in a brute-force $O(n^2)$ fashion, where each pixel is compared to its expected result in the corresponding formula. In addition to the formulae, three dimensional visuals are generated in Wolfram Mathematica for visual validation of library results. The aggregation results are compared to the results of simpler aggregation algorithms that are non-vectorized and use brute force. These are compared using a built in NumPy function. These tests are covered in Section 3.

The interesting part of this library is the ability to combine multiple different analyses into a single image. As example applications we use k-means clustering, a way of clustering data into discrete groups, and pairplots, a way of plotting multiple datasets against each other in a single image. Although the library already contains a fair amount of analyses to choose from, more may still be added. Because of the library's open source nature, anybody can contribute a new analysis utilizing the sliding window algorithm. Even algorithms without a need for aggregations can be added, such as the already present NDVI function. Results of these combinations are covered in Section 4.

1.3. Software Design

When developing an open source library, careful consideration needs to be taken in the decision of the chosen programming paradigm, language, and libraries. According to Castro [3], the decision of software design should never really be about the programming language. It is much more mentally taxing for developers to switch design paradigms than programming languages. It is the design paradigm that ultimately affects the architecture of the software. For this thesis, the authors have opted for an imperative paradigm. Imperative programming languages are more ubiquitous, with the added benefit of packaging data with its associated algorithms. These features allow our library to be more approachable and logically coherent. To aid in the readability of our library, the Python programming language was chosen. Python is famous for being an easy to use and easy to learn langauge. It is extremely popular among newer developers and data scientists. In addition, it has the open source library NumPy, which is a very popular library in the scientific computing community. NumPy is actually an open source academic project [9], like this thesis is. NumPy is written in C, with bindings to Python. This allows us to utilize the expressiveness and readability of Python combined with the high performance of C.

A misconception among inexperienced software engineers is that all well designed code is object-oriented. As Stroustrup [16] has pointed out, this is untrue. This misconception only goes to show what sorts of paradigms young developers are being taught and the prevalence of object-oriented programming in the software space. This is not a bad thing of course. The principles of object-orientation, such as polymorphism and data abstraction, are great boons in some designs. They enable separation of concerns and an increase in abstract thinking. The only caveat is that these qualities are not suited for all problems. Stroustrup outlines four different programming paradigms that are imperative: procedural, data hiding, data abstraction, and object-oriented. This thesis uses data abstraction. This paradigm involves breaking up code into functions and user defined types. User defined types are software modules where operations such as operators, initializations, and deallocations can be defined. This helps to manage the lifecycle of memory and conceptualizes blocks of code into objects. Actual object-oriented programming involves finding commonalities among user defined types and putting them into hierarchies. Types higher up in the hierarchy can be used to generalize different operations on lower types. For examples, in a graphical program, a circle can inherit from a shape object and is able to use the ”draw” function defined in the shape object. The reason we are not using object-oriented programming is because adding a hierarchy of types would add unnecessary complexity to our library.

2. CONCEPTS

2.1. Sliding Window

The sliding window aggregation algorithm is done over windows of increasing sizes, reusing aggregates from previous steps. For each step, the window size will be 2^n pixels square, where n is the current aggregation number. The corners of a sub window $2^{n-1}+1$ pixels square are aggregated. The corners of the sub window represent non-overlapping aggregates from the previous step. These combine to create the whole aggregate of the window in the current aggregation step. The window is doubled in scale every step, resulting in $O(\log(n))$ time complexity. The first three aggregations of this process can be seen in Figure 2.1.

For aggregation number one, corner pixels of the window $2^{1-1} + 1 = 2$ pixels square are aggregated. These form the aggregate of a window $2^1 = 2$ pixels square. This can be seen in the top left corner of Figure 2.1. The next iteration we add the corners of the $2^{2-1}+1 = 3$ pixel square window. This includes the aggregate from the last step, plus the three immediate non-overlapping windows next to it. These form the aggregate of the $2^2 = 4$ pixel square window. This can be seen in the bottom left corner of Figure 2.1. This process repeats itself until the desired amount of aggregations is acheived.

This same aggregation stategy can be used for a variety of algorithms in the library. Using this algorithm to sum all pixels is useful for correlations and fractal dimensions, where a sum is all that is required. Operations other than summing all four pixels each iteration can be used. For example, pixels on the $-x$ axis can be subtracted while pixels on the $+x$ axis are added. The same operation can be done for the y axis. These and other special aggregates are used in the topographical calculations later in this thesis.

2.2. Software Architecture

Well designed software is decoupled and human-readable. These were the two concepts at the forefront of the author's minds when designing this software library. The most obvious form of decoupling is the application programming interface (API). An API is the interface between two different pieces of software, it is how they communicate. When a piece of software interacts with another using an API, it doens't care how the backend of the API is implemented, or how it stores

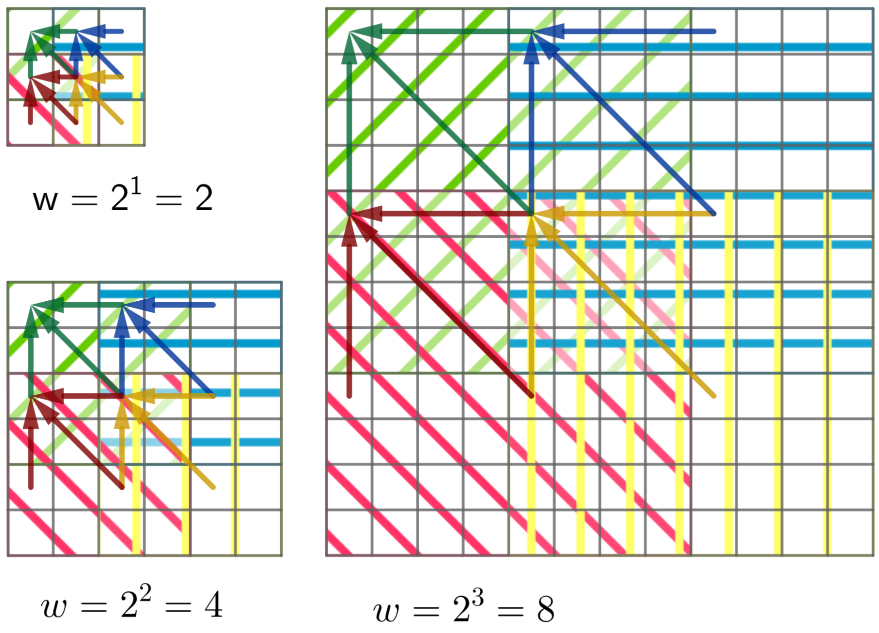


Figure 2.1. The first three iterations of the sliding window aggregation algorithm. Each iterations has a sliding window size of 2^n for the n^{th} aggregation. Arrows represent individual pixels being aggregated together. The aggregates of the shaded regions are stored in the top left corner of the region.

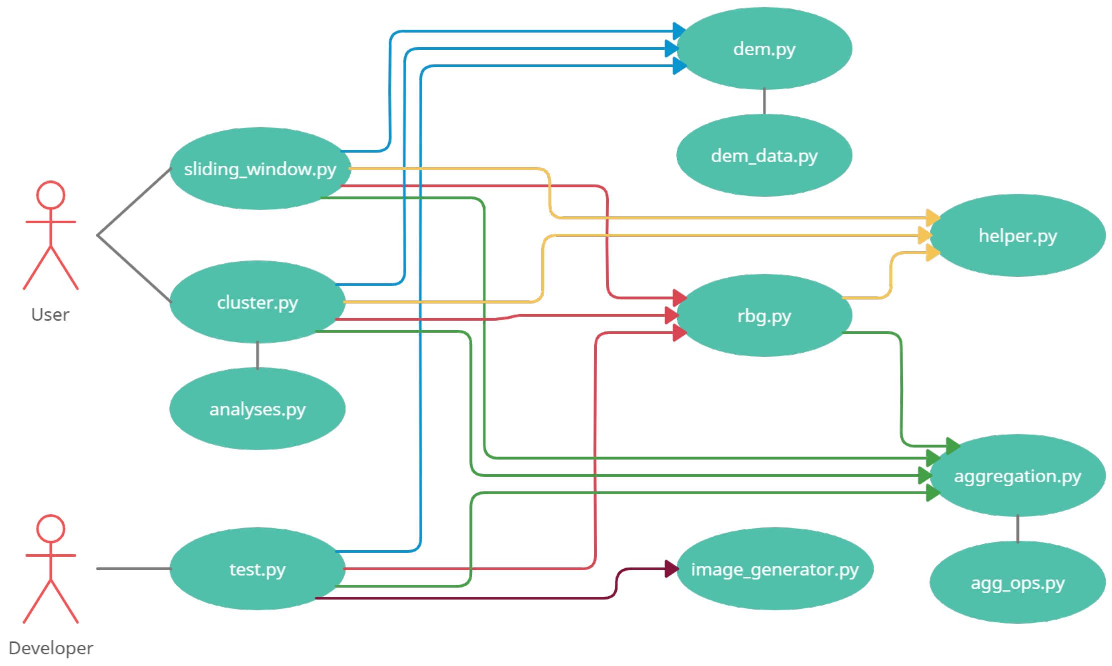


Figure 2.2. Diagram representing how all of the files in the library utilize each other. Files at the tails of arrows are importing the files at the heads of the arrows. Lines with no arrow head represent related files that are generally imported together. The actors represent those who use the library and what files they generally interface with.

its data. All the requesting software cares about is if the API does what it says it will do. This library is decoupled and broken up into 7 main Python files, which are covered in detail later in this section. A diagram describing how the files import each other can be seen in Figure 2.2.

To make the library human readable, much care was taking in the nomenclature of identifiers. Logical structure and nomenclature took presedence over verbose comments. This helps to protect the library from future contributions that may forget to update the comments. In addition, consistency with theoretical results will also help in the creation of human readable, expressive code. Code resulting from mathematical derivations can result in confusion for the reader. With a resemblance to theoretical results, a reader can refer to the research where the equations were derived to get a better understanding of the results. With the non-trivial nature of the sliding window computations, this is likely to be the most important step in readable code.

2.2.1. sliding_window.py

The sliding window class is at the highest abstraction level of the library. It is designed to be the main way users interface with the library. The only parameters that the class requires are the file name and the width/height in meters per pixels. In an ideal situation, the class would only require the file name, but at the time of writing the class is unable to dynamically determine the width and height. This information does exist in the metadata, but is not guaranteed to be in the correct units. Metadata of GIS data can also be in angular degrees.

This class interfaces with many of the classes at lower levels of abstraction and combines them all together. These include: rbg.py, dem.py, helper.py. The user merely has to specify an operation and a spectral band, and this class takes care of the band extraction, calculations, and saving to disk. Additionally, options have been added to save a lower resolution plot of the output, to convert the output datatype, and to specify the work datatype or output datatype. The python library MatPlotLib is used to generate the lower resolution images. These enable quick and easy viewing or sharing of results, when further analysis of the image isn't required.

2.2.2. aggregation.py

This file contains all of the aggregation methods. These aggregation methods do not require state, and thus are not written in a class. They are simply exported from the file. Included among the functions are vectorized versions, which utilize Single Instruction Multiple Data (SIMD) CPU architecture, and normal brute-force operations. As the vectorized versions are not entirely intuitive to understand, they have been written adjacent to the brute-force operations which follow a much more logical structure but are far less efficient. These two methods are compared directly in the test class for validation.

The vectorized aggregation methods aggregated pixels in a w long square area, where $w = 2^n$ and n is the number of aggregations. The resulting aggregation is saved in the top left pixel of the aggregated area. As such, aggregation outputs an image of the same resolution, containing aggregated values, and truncated by $2^n - 1$ units on the $+x$ and $+y$ axes for pixels whose aggregates existed off the image. When saving these aggregated images they must be shifted in their metadata by $\frac{2^n-1}{2}$ pixels. Shifting occurs to put the aggregated values in the center of the aggregation areas, instead of in the top left corner.

This class could not simply accept an aggregation operations and a number of of aggregations. This would not allow for mixing and matching of the aggregation operations. The functions also requires knowledge of the number of previous aggregations, to enable the function to calcuate aggregations in chunks instead of all at once. The number of previous aggregations sets the window size that is used for aggregation. If this is set incorrectly the function has undefined behavior.

As well as having the normal aggregation method, there is also an aggregation method for DEM files. Per the DEM algorithm the following values must be aggregated: z, xz, yz, xxz, yyz, xyz. These represent the average values of the corresponding product. The window is assumed to be centered at 0, z being the image value, x being the x coordinate of the pixel, and y being the y coordinate of the pixel. All of these values, including the number of aggregations, are stored in an auxilliary data structure called *Dem_data*. Thus, the only arguments required are a *Dem_data* object and the number of desired additional aggregations. Like with the normal aggregation method, these average values calculated also have appropriate brute-force testing methods that behave identically.

2.2.2.1. agg_ops.py

Contained in agg_ops.py is an enum describing the different available aggregation operations implemented in aggregation.py. These include: add_all, add_bottom, add_right, add_main_diag, maximum, and minimum. The aggregation function only add 4 values at a time, helping to describe its $O(\log(n))$ performance. With this in mind, it can be seen that the operations are relatively self explanatory. The operation add_all adds all values, add_bottom adds the values on the $+y$ axis, and so on.

This file exists by itself instead of within the aggregation.py file to allow it to be used without aggregation.py. For example, if an operation were to be added to the SlidingWindow class which allowed the user to specify the aggregation method, then this file could be imported to specify the operation.

2.2.3. rbg.py

The lower level analysis operations that can be performed on multispectral bands exist in rbg.py. This file does not require state and only exports functions. These functions are not only for images with red, green, and blue bands. The name rbg.py is merely a convenient name. These operations can be used on any spectral image. In fact, this file has no concept of spectral bands, that only exists in sliding_window.py. Any NumPy arrays can be passed into theses functions given that they all have the same datatypes and shapes. This file also attempts to maintain the datatypes of the passed in arrays. This is to make it very transparent who is manipulating the representation of numbers being used. The user is expected to pass in the appropriate data types to avoid overflow. If an 8-bit array is passed in and numbers are aggregated above the maximum value of 255, overflow will occur.

Operations included in this file are: ndvi, binary, regression, pearson, fractal, and fractal_3d. Two of these operation do not actually use the sliding window algorithm: ndvi and binary. The goal of this software library is not to include all possible applications of the sliding window algorithm, it is to include a variety of operations in one package so they may be combined and compared. The operations using the sliding window algorithm are all aggregated over a 2^n square area, with n being the number of aggregations. Thus each pixel in the resulting image will describe a 2^n square area in the original image.

2.2.3.1. NDVI

The ndvi function, or Normalized Difference Vegetation Index, is a common analysis used in GIS to asses the presence of vegetation. It is a simple difference between the red and infrared bands, not requiring the sliding window algorithm. Since there is no concept of spectral bands in rbg.py, the user is expected to pass in the red and infrared bands as NumPy arrays to get the expected results.

2.2.3.2. Binary

This function is relatively self explanatory. It takes in an array with a range of values, and maps them to two values. Using a threshold parameter, it can determine how to map the array values. The threshold parameter is a percent of the existing values in the image. Our threshold value will be $((threshold * (max - min)) + min)$. Values less than this threshold value will be zero and values greater than or equal to it will become the maximum value of the desired data type. The maximum and minimum values present in the image are used to ensure the entire image doesn't get mapped to the exact same value.

2.2.3.3. Correllations

The regression and pearson are common statistical correlation tools. A statistical regression function will commonly return a linear polynomial, a best fit line describing the relationship between the two sets of numbers. The regression present in this library uses the sliding window algorithm to aggregate values and returns the slope of the regression line. These slope values are used to generate the resulting image, describing the relationship between the specified spectral bands. The pearson corrleation is very similar, although it returns a value in the range $[-1, 1]$. A value of -1 respresents a perfect indirect relationionship, a 0 represents zero correlation, and a 1 represents a perfect direct relationship.

2.2.3.4. Fractal Dimension

The degree of self similarity of a geometric object can be described by its fractal dimension. That is, in more colloquial terms, the degree of "roughness" of a geometric shape. If a rough shape continues to be rough as it is scaled to larger sizes, then it is more likely to have a non-integer fractal dimension. In general, geometric shapes are considered to have integer dimensions, zero for a point, one for a line, and so on. Fractal dimensions describe the non-integer dimension, if they exist. For example, the prototypical perfect fractal, the Sierpiński triangle, has a fractal dimension of 1.585. This function returns an image where each pixel is a fractal dimension. The 3D variant of this function does the same, with the exception that array values are treated as elevation and fractal dimensions are computer over a 2^n cubed volume instead of a square area.

2.2.4. dem.py

The second lower level analysis file is dem.py, working very similarly to rbg.py. As the name implies, it is intended to be used with DEMs. It still has no state, but requires a *Dem_data* object to hold its working data, which does have state. The passed in *Dem_data* object contains all of the data required for array aggregation, the other parameters required for many functions in this file are *pixel_width* and *pixel_height*. For DEMs, the elevation and the pixel width/height are often not in the same units. For this reason, these values must be passed in for all but the aspect function. This file contains three functions: slope, aspect, and curvature. Just like in rbg.py, each resulting pixel from these functions describes a 2^n square area, with n being the number of aggregations. A polynomial is fit to the 2^n square area and calculus derived formulae are then applied.

2.2.4.1. Slope

This method returns an image where each pixel is the slope of the corresponding location in the original image. In the SlidingWindow class, a slope angle operation is also supported, which simply takes the arc tangent of the result from this function.

2.2.4.2. Aspect

This method doesn't require pixel width or height because it is only a direction. Ouput from this method describes the direction of steepest descent, i.e. the direction opposite to the slope. This is denoted following GIS conventions. Aspect is defined as the degrees in radians from north. Due to the $+y$ axis pointing down in raster images, the positive direction for radians is clockwise, instead of the normal counter-clockwise direction. For example, if the direction of steepest descent is due east, the aspect would be $\frac{\pi}{2}$ radians.

2.2.4.3. Curvature

To understand curvature, we much first understand that the slope is the first derivative in the direction of steepest descent. Several different types of curvature exist: profile, planform, and standard. Profile is the second derivative in the direction of steepest descent. Planform is the second derivative in the direction perpendicular to the steepest descent. Standard is the average of profile and planform. This library only contains a function for standard curvature.

2.2.4.4. dem_data.py

The container for the working data of dem.py is found in dem_data.py. It contains arrays for the aggregated averages of z, xz, yz, xxz, yyz, and xyz. It automatically ensures that all of these arrays are the same data type and shape. It also has a value to keep track of the number of aggregations. These values can be set directly, but the preferred way is to let the *aggregate_dem* function in aggregations.py do it for us. Lastly, this class offers the ability to export and import working data using numpy's archiving methods: *numpy.savez* and *numpy.load*. Since aggregating all these arrays takes a non-trivial amount of time to do, exporting and importing are very helpful methods when working with DEMs.

2.2.5. cluster.py

The purpose of cluster.py is to combine multiple analyses into a single image. This is done by specifying multiple different analyses, bands, and aggregations numbers. One of the main purposes of this library is to enable the combined usage of many different statistical methods. This file is really the cornerstone of such a feature. Cluster.py currently contains two functions, *gen_clustered_img* and *gen_pairplot_img*. Many different arguments can be passed to these function to specify their behavior. Results of both of these operations can be see in Section 4.

Our first function is *gen_clustered_img*. In this function, m dimensional vectors are clustered, where m is the number of analyses. There will be one vector for each pixel in the output image. For a vector, each index will be populated with the corresponding output from one of our specified analyses.

For *gen_clustered_img*, our first argument is the file path, the path to the image being analyzed. Following that are three arguments named *analyses*, *num_aggres*, and *bands*. These three are Python lists containing information about the different analyses to be clustered. The *analyses* parameter contains values from the analyses enum, which is convered in the following subsection. The *num_aggre* parameter contains the number of aggregations. The *bands* parameter contains the bands to extract from the image to use as input to the analyses. These can either be a single band, like for DEM methods, or a list of 2 bands, like for the correlation methods.

The rest of the options are more fine grained control of the clustering operation. There is *num_clusters*, the number of groups to cluster the vectors into. Next there is the *sub_image_* *size*, which allows us to analyze only a small sub image of the original, to speed up computations. Currently, the default for this value is 256. After that is *sub_image_start*, which is a python list of size two. This specifies the coordinates of the top left corner our sub image. If the sub image extends beyond the images bounds, an error will be thrown. Following that is the *map_width_to_meters* and *map_height_to_meters* for DEM operations, and then the *output_file_path*. Lastly, we have the *average_cluster_values* boolean parameter. This tells the function to create output values that are either the cluster number, or the average of the dimensions of the cluster center vector.

The second function, *gen_pairplot_img* , is very similar to the first. Instead of clustering the analysis results, they are plotted against each other. Each analysis will be plotted on the x axis in a column and the y axis in a row. Along the diagonal are bar graphs of a single analysis. The arguments to this function are identical to *gen_cluster_img* except for the fact that the *average_cluster_values* and *num_clusters* arguments are omitted, as there are no cluster values.

As a result of these functions accepting analyses with a range of aggregation numbers, many analyses will have to be truncated to reach a standard size. The resulting image of the functions will be the same size as the analysis with the largest number of aggregations, which has the largest amount of truncated pixels. $2^n - 1$ truncated pixels to be exact, with n being the number of aggregations. Every other analysis will have to have $2^i - 2^j$ additional pixels truncated, where i is the maximum amount of aggregations of any analysis, and j is the number of aggregations of the analysis in question.

As a results of the cluster.py functions using sub images and truncating pixels with the aggregation function, it is diffictult to compare results back to the original image. To remedy this, an "adjusted" image is created. It is not possible to simple truncate pixels off of the original image to create our adjusted image. That would require us to remove a half pixel from each side. This is due to the fact that $2^n - 1$ pixels are truncated when aggregating. We would need to remove $\frac{2^n-1}{2}$ pixels from each side, resulting in a half pixel because $2^n - 1$ is an odd number. Additionally, it is not wise to simply aggregate the original image to match the output. This could result in a very blurry image if a sufficiently small sub image is used. The solution is to only aggregate 4 pixels together, resulting in $\frac{2^1-1}{2} = \frac{1}{2}$ pixels being removed from each side. Then $\frac{2^n-2^1}{2}$ pixels can be truncated from each side, which is guarenteed to be an integer because $2^n - 2$ is an even number.

2.2.5.1. analyses.py

The analyses enum contains eight options: ndvi, regression, pearson, fractal, fractal_3d, slope, aspect, and standard. Each of these methods corresponds to one of the functions in either rbg.py or dem.py. If new additions to these files are made, more options will have to be added to this enum and corresponding logic will have to be added to cluster.py.

2.2.6. test.py

This library is tested using unit tests with Python's standard *unittest* package. Methods have been written to test aggregation and DEM functions. Fractal functions have yet to be tested. These tests can easily be run by using Python to run the test.py class: *python3 /path/to/test.py* in Linux or *python \path\to\test.py* for Windows.

For testing the aggregation file, the brute-force functions are written in aggregation.py. All that is done in the test.py file is a comparison of the results of the vectorized functions and the brute-force funtions. The vectorized functions use Denton's sliding window algorithm, and the brute-force operations use a simple operation with $O(n^2)$ time complexity.

The testing methods for the DEM functions are slightly less transparent. These tests use analytical results derived in Wolfram Mathematica. Results are obtained from the typical vectorized DEM functions. Next, the analytically derived functions are used to determine the correct value for every pixel. Again with $O(n^2)$ time complexity. This testing is covered more in depth in Section 3.

2.2.6.1. image_generator.py

For input to our testing methods we can use results from the image_generator.py file. This file has the ability to make several different algorithmic images. Arguably the most useful is the Gaussian hill function. This function, along with the functions using the Wolfram Mathematica analytical results, are used to test the functions in the dem.py file. The analytical Wolfram functions also exist in this file, they simply return a single point instead of generating a whole image. For all the algorithmic image generation functions, size can be specified. The values *mu* (μ) and *sigma* (σ) can also be specified for the Gaussian functions. These values describe the Gaussian hill, its standard deviation and offset respectively.

2.2.7. helper.py

For functions that don't quite belong in any one file, or are used by multiple different files, this auxilliary file was made. It contains such fuctions as: *dtype_max_min*, *create_tif*, and *arr_dtype_conversion*.

The tiff creation function helps to automate the process of creating tiffs from multiple arrays and with modified profile information. In rasterio, the GDAL library we are using, the profile of an image is all of the metadata. This includes information such as an image's location on the globe. The profile object primarily has to be modified to account for the truncated pixels after aggregation. This is why the *num_aggre* parameter on this function exits. This function can also generate multi-spectral images by accepting a Python list of numpy arrays. Each element of the Python list will become a band in the resulting image.

The image conversion algorithm used in the *arr_dtype_conversion* function is fairly simple. It requires a *low_bound* and a *high_bound* parameter. If these are not supplied, they are set to the minimum and maximum values of the array, respectively. The input array is then mapped from the range $[low\_bound, high\_bound]$ to the range $[0, newDataType.max]$. A copy of the array is returned, converted to the new data type and with its values mapped as previously mentioned.

2.3. Topographical Calculations

Using DEM files, we can fit polynomials to the topography represented in the file. These polynomials are written in terms of elevation-data aggregates and xy-coordinate aggregates. They will be used in Section 2.4 to derive our mathematical formulae to calculate slope, aspect, and curvature. A brief review of the methods used to calculate these polynomals will follow in this section.

2.3.1. Slope

As discussed by Denton et. al. [6] the slope computations are as follows. The polynomial we are fitting to the DEM is of the following form:

$$z_1(x,y) = \begin{pmatrix} b_0 & b_1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + c_s = b_0x + b_1y + c_s \tag{2.1}$$

We minimize the squared error from our image values to fit the polynomial.

$$\langle (z - z_1(x,y))^2 \rangle = \langle (z - xb_0 - yb_1 - c_s)^2 \rangle \tag{2.2}$$

Where $\langle v \rangle$ represents an average for any variable v.

We take the partial derivatives with respect to b_0, b_1, and c_s, then set them to zero to minimize. With our resulting equations, many of the terms can be cancelled out with the assumption that our image is centered on the origin. For example, the $\langle x \rangle$ term will be zero, equal parts of the image exist on the $+x$ axis and the $-x$ axis. Our resulting equations are the following:

$$0 = \langle xz \rangle - \langle x^2 \rangle b_0 \qquad 0 = \langle yz \rangle - \langle y^2 \rangle b_1 \qquad 0 = \langle z \rangle - c_s \tag{2.3}$$

Solving these we get:

$$b_0 = \frac{\langle xz \rangle}{\langle x^2 \rangle} \qquad b_1 = \frac{\langle yz \rangle}{\langle y^2 \rangle} \qquad c_s = \langle z \rangle \tag{2.4}$$

Again, assuming our grid is centered on the origin, we can see that $\langle x^2 \rangle = \langle y^2 \rangle$. Using a grid of pixels, each of our pixels will have coordinates that are between whole numbers, e.g. $\frac{1}{2}$ or $\frac{3}{2}$. We can see that the resulting summation for $\langle x^2 \rangle$ with w window size is:

$$\langle x^2 \rangle \quad = \quad \langle y^2 \rangle \quad = \quad \frac{1}{w^2} 2w \sum_{k=1}^{\frac{w}{2}} (k - \frac{1}{2})^2 \quad = \quad \frac{2}{w} \left(\sum_{k=1}^{\frac{w}{2}} k^2 - \sum_{k=1}^{\frac{w}{2}} k + \frac{w}{8} \right) \tag{2.5}$$

To solve the summations we can use Gauss's Identity and Sum of Squares:

$$\sum_{k=1}^{\frac{w}{2}} k = \frac{\frac{w}{2}(\frac{w}{2}+1)}{2} = \frac{w^2}{8} + \frac{w}{4} \tag{2.6}$$

$$\sum_{k=1}^{\frac{w}{2}} k^2 = \frac{\frac{w}{2}(\frac{w}{2}+1)(2\frac{w}{2}+1)}{6} = \frac{w^3}{24} + \frac{w^2}{8} + \frac{w}{12} \tag{2.7}$$

Solving for $\langle x^2 \rangle$ we have:

$$\langle x^2 \rangle = \frac{2}{w} \left(\left(\frac{w^3}{24} + \frac{w^2}{8} + \frac{w}{12} \right) - \left(\frac{w^2}{8} + \frac{w}{4} \right) + \frac{w}{8} \right) = \frac{w^2 - 1}{12} \tag{2.8}$$

Now we can construct our polynomial that is fit to the DEM file in terms of $\langle z\rangle$, $\langle xz\rangle$, $\langle yz\rangle$, and w. These aggregates are computed in the aggregation.py file and are coverered in more depth in Denton's paper on Slope computations [6].

2.3.2. Curvature

A similar process is followed for curvature, using a quadratic polynomial. Our quadratic formula looks like the following:

$$\begin{aligned} z_2(x,y) &= \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} a00 & a10 \\ a10 & a11 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} b_{c0} & b_{c1} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + c_c \\ &= a_{00}x^2 + 2a_{10}xy + a_{11}y^2 + b_{c0}x + b_{c1}y + c_c \end{aligned} \tag{2.9}$$

We minimize the squared error from our image values to fit the polynomial.

$$\langle (z - z_2(x,y))^2\rangle = \langle (z - a_{00}x^2 - 2a_{10}xy - a_{11}y^2 - b_{c0}x - b_{c1}y - c_c)^2\rangle \tag{2.10}$$

Using partial derivatives again, and canceling out symmetric variables, we get:

$$\begin{aligned} \langle x^2 z\rangle &= \langle x^4\rangle a_{00} + \langle x^2y^2\rangle a_{11} + \langle x^2\rangle c_c \\ \langle xyz\rangle &= 2\langle x^2y^2\rangle a_{10} \\ \langle y^2 z\rangle &= \langle x^2y^2\rangle a_{00} + \langle y^4\rangle a_{11} + \langle y^2\rangle c_c \\ \langle xz\rangle &= \langle x^2\rangle b_{c0} \\ \langle yz\rangle &= \langle y^2\rangle b_{c1} \\ \langle z\rangle &= \langle x^2\rangle a_{00} + \langle y^2\rangle a_{11} + c_c \end{aligned} \tag{2.11}$$

We use our symmetrical grid to assert that $\langle x^2\rangle = \langle y^2\rangle$ and $\langle x^2y^2\rangle = \langle x^2\rangle^2$ are true. Next we solve for our desired variables to form our polynomial.

$$\begin{aligned}
a_{00} &= \frac{\langle x^2z\rangle - \langle x^2\rangle\langle z\rangle}{\langle x^4\rangle - \langle x^2\rangle^2}\\
a_{10} &= \frac{\langle xyz\rangle}{2\langle x^2\rangle^2}\\
a_{11} &= \frac{\langle y^2z\rangle - \langle x^2\rangle\langle z\rangle}{\langle x^4\rangle - \langle x^2\rangle^2}\\
b_{c0} &= \frac{\langle xz\rangle}{\langle x^2\rangle} = b_0\\
b_{c1} &= \frac{\langle yz\rangle}{\langle x^2\rangle} = b_1\\
c_c &= \langle z\rangle - \frac{\langle x^2\rangle\langle x^2z\rangle + \langle x^2\rangle\langle y^2z\rangle - 2\langle x^2\rangle^2\langle z\rangle}{\langle x^4\rangle - \langle x^2\rangle^2}
\end{aligned} \tag{2.12}$$

The only term we have left to solve for is $\langle x^4\rangle$. To solve this we can follow a process similar to what was done for the slope polynomial:

$$\langle x^4\rangle \quad = \quad \langle y^4\rangle \quad = \quad \frac{1}{w^2}2w\sum_{k=1}^{\frac{w}{2}}(k-\frac{1}{2})^4 \quad = \quad \frac{2}{w}\left(\sum_{k=1}^{\frac{w}{2}}k^4 - \sum_{k=1}^{\frac{w}{2}}2k^3 + \sum_{k=1}^{\frac{w}{2}}\frac{3k^2}{2} - \sum_{k=1}^{\frac{w}{2}}\frac{k}{2} + \frac{w}{32}\right) \tag{2.13}$$

Using more power summations:

$$\sum_{k=1}^{\frac{w}{2}}k^3 = \frac{(\frac{w}{2})^4}{4} + \frac{(\frac{w}{2})^3}{2} + \frac{(\frac{w}{2})^2}{4} = \frac{w^4}{64} + \frac{w^3}{16} + \frac{w^2}{16} \tag{2.14}$$

$$\sum_{k=1}^{\frac{w}{2}}k^4 = \frac{(\frac{w}{2})^5}{5} + \frac{(\frac{w}{2})^4}{2} + \frac{(\frac{w}{2})^3}{3} - \frac{\frac{w}{2}}{30} = \frac{w^5}{160} + \frac{w^4}{32} + \frac{w^3}{24} - \frac{w}{60} \tag{2.15}$$

Adding it all together:

$$\begin{aligned}
\langle x^4 \rangle &= \frac{2}{w}\left(\sum_{k=1}^{\frac{w}{2}} k^4 - \sum_{k=1}^{\frac{w}{2}} 2k^3 + \sum_{k=1}^{\frac{w}{2}} \frac{3k^2}{2} - \sum_{k=1}^{\frac{w}{2}} \frac{k}{2} + \frac{w}{32}\right) \\
&= \frac{2}{w}\left((\frac{w^5}{160} + \frac{w^4}{32} + \frac{w^3}{24} - \frac{w}{60}) - 2(\frac{w^4}{64} + \frac{w^3}{16} + \frac{w^2}{16}) + \frac{3}{2}(\frac{w^3}{24} + \frac{w^2}{8} + \frac{w}{12}) - \frac{1}{2}(\frac{w^2}{8} + \frac{w}{4}) + (\frac{w}{32})\right) \\
&= \frac{3w^4 - 10w^2 + 7}{240}
\end{aligned} \tag{2.16}$$

Now we can construct our second polynomial that is fit to the DEM file in terms of $\langle z \rangle$, $\langle xz \rangle$, $\langle yz \rangle$, $\langle x^2 z \rangle$, $\langle y^2 z \rangle$, $\langle xyz \rangle$, and w. These aggregates are computed in the aggregation.py file.

2.4. DEM Formula Generation

Using the polynomials that were just derived in Sections 2.3.2 and 2.3.1, we will use some calculus to derive our formulas used in the library. Wolfram Mathematica was used to derive our results, the code snippet can be seen in Figure 2.3. In Mathematica, we are deriving results for slope, aspect, and standard curvature. This will include variants with map unit normalization and ones without. We will also be comparing the results from the linear and the quadratic polynomial equations.

Our linear equation is labeled g, and our quadratic is f. Our computations are fairly straight forward. Taking the partial derivatives with respect to x and y. We then normalize and negate those values to get our direction of steepest descent. We take the derivative of g in the direction of steepest descent to get our slope equation.

For our aspect equation, we must use a special form of the arctangent function that accepts two parameters. These parameters are x and y respectively, and this function uses those values to return us the arctangent result in the correct quadrant. This function will return us the aspect starting at due East and moving in a counter-clockwise direction. We want the aspect starting from due North and moving in a clockwise direction. First we add $\frac{\pi}{2}$ to move our starting point 90°. Our y values are actually already flipped because we are working with raster data, where $+y$ is in the downward direction. This causes our aspect to be calculated in a clockwise direction. Lastly, we use the modulo operator to wrap values back to zero.

To normalize with our map units, we need to get the length in meters of moving one unit in the direction of steepest descent. To do this we require the length in meters between adjacent pixels on the x axis and the length in meters between adjacent pixels on the y axis. These values can be derived using the conversion values passed to sliding_window.py, *map_width_to_meters* and *map_height_to_meters*, as well as the transform data from the rasterio image. Now to obtain our length in meters in the direction of steepest descent, we multiple our x slope direction by *map_width_to_meters* and our y slope direction by *map_height_to_meters*. After this we use Pythagoreans theorem and we are done. We will divide our slope and curvature computations by this value to normalize them to map meters. This operation is not required with the aspect function because it only describes a direction.

A process identical to g's is followed for f. In addition, we will be taking second derivatives in the direction of steepest descent and the direction perpendicular to steepest descent. These two values are the profile and planform equations, respectively. Averaging these values gives us the equation we want, the standard curvature.

When viewing our results at the bottom of the code snippet in Figure 2.3, we set our x and y values to zero. This is because our aggregates are designed to describe a point at the origin of the aggregation. Thus we only care when $x = 0$ and $y = 0$. Viewing our results we can see that our linear and quadratic equations are identical for slope and aspect. We display these results in Equations 2.17 through 2.21, ignoring the duplicates. These are the equations used in the sliding window library, using our previously derived symbols such as b_0 and a_{00}.

$$-\frac{\sqrt{b0^2 + b1^2}}{\sqrt{\frac{b1^2 mapHeight^2 + b0^2 mapWidth^2}{b0^2 + b1^2}}} \tag{2.17}$$

$$-\sqrt{b0^2 + b1^2} \tag{2.18}$$

$$(\frac{\pi}{2} + \operatorname{atan2}(b0, b1)) \mod 2\pi \tag{2.19}$$

$$\frac{a00 + a11}{\sqrt{\frac{b1^2 mapHeight^2 + b0^2 mapWidth^2}{b0^2 + b1^2}}} \tag{2.20}$$

$$a00 + a11 \tag{2.21}$$

2.5. Installation

The project exists in a GitHub repository and it will exist in the Python Package Index. On the web, it can be found at this GitHub URL: *https://github.com/amdenton/SlidingWindows*. When the project is published to the Python Package Index, it will also be available at this PyPi URL: *https://pypi.org/project/name_to_be_determined*. These URLs just describe the library. To install and use it, Python3 and pip will have to be installed. If GitHub is preferred, git will need to be installed as well. The pip installation is preferred for simple usage, as it makes the package available everywhere in the Python environment. The Github installation is useful for modifying the source code, but the source files must be imported directly. If we desire to contribute source code changes to the original project, we will need to fork and send a pull request to the GitHub repository. This process of contributing is not outlined in this thesis. With either installation process, we must follow special installation instructions due to a dependency of the project called *Rasterio*.

2.5.1. Virtual Environment

When using git, the author uses a virtual environment to encapsulate the Python environment for development purposes. When Python packages are installed after such an environment is activated, the packages will only exist in that virtual environment and will only be accesible when it is activated. They will have no affect on the rest of the system. Such an environment is activated and deactivated with generated scripts, which differ between operating systems. To create our environment we will use the Python package *virtualenv*.

- Activation
  - Windows
    - $ cd \path\to\desired\directory\
    - $ python -m pip install –upgrade pip

    * $ python -m pip install virtualenv
    * $ python -m virtualenv *EnvironmentName*
    * $.\*EnvironmentName*\\Scripts\\activate.bat
  - Linux
    * $ cd /path/to/desired/directory/
    * $ python3 -m pip install –upgrade pip
    * $ python3 -m pip install virtualenv
    * $ python3 -m venv *EnvironmentName*
    * $ source *EnvironmentName*/bin/activate
- Deactivation
  - $ deactivate

2.5.2. Rasterio Installation

Rasterio allows us to access geospatial data of raster images. As such, we need to install the Geospatial Data Abstraction Library (GDAL), which is a C library and not readily available on pip. As an alternative to globally installing GDAL on Windows, we will be using some high quality unofficial binaries created by Christoph Gohlke. This will allow us to manage GDAL with the virtual environment. For Linux, it will be easier for us to globally install GDAL from the source distribution. This will not be managed by the virtual environment and will have to be manually removed later. According to the installation instructions on Rasterio's official docs website: *https://rasterio.readthedocs.io/en/latest/installation.html*, Rasterio can be installed with the following steps:

- Windows
  - Download GDAL binaries from *https://www.lfd.uci.edu/~gohlke/pythonlibs/#gdal*
  - Download Rasterio binaries from *https://www.lfd.uci.edu/~gohlke/pythonlibs/#rasterio*
  - $ python -m pip install –upgrade pip
  - $ python -m pip install \\path\\to\\GDAL\\binaries.whl

  - $ python -m pip install \path\to\Rasterio\binaries.whl

- Linux (Ubuntu)
  - $ sudo add-apt-repository ppa:ubuntugis/ppa
  - $ sudo apt-get update
  - $ sudo apt-get install python-numpy gdal-bin libgdal-dev
  - $ python3 -m pip install rasterio

2.5.3. SlidingWindow Installation

Github is currently our only option for installing the Sliding Window library. It is planned to be released on the Python Package Index in the future to make a pip installation available as well. Pip is the easier option, and it is preferred for the typical user who has no need to modify the code or contribute new code. The Rasterio intructions still need to be followed with the pip installation, but only one commmand is required after that. In addition, pip allows the package to be available from everywhere in the environment with a simple import statement such as *import packagename.*

Github is the preferred installation method for the curious reader who desires to read through the algorithms in the library, or contribute new ones. Since pip will not be directly managing our dependencies for us, it is helpful to create a virtual environment to encapsulate them. Although, then the library will only be available for use when the environment is activated. A typical git clone operation happens first, then a rasterio installation, followed by a pip installation of the requirements file. The requirements file enumerates all of the dependencies of the package. When using the package with git it will have to be imported with a path to the source files, e.g. *import /path/to/package/source/file.*

- pip *(not available as of writing)*
  - Install Rasterio, see Section 2.5.2
  - Install SlidingWindow
    - ∗ Windows

      $ python -m pip install *name_to_be_determined*

    * Linux

      $ python3 -m pip install *name_to_be_determined*

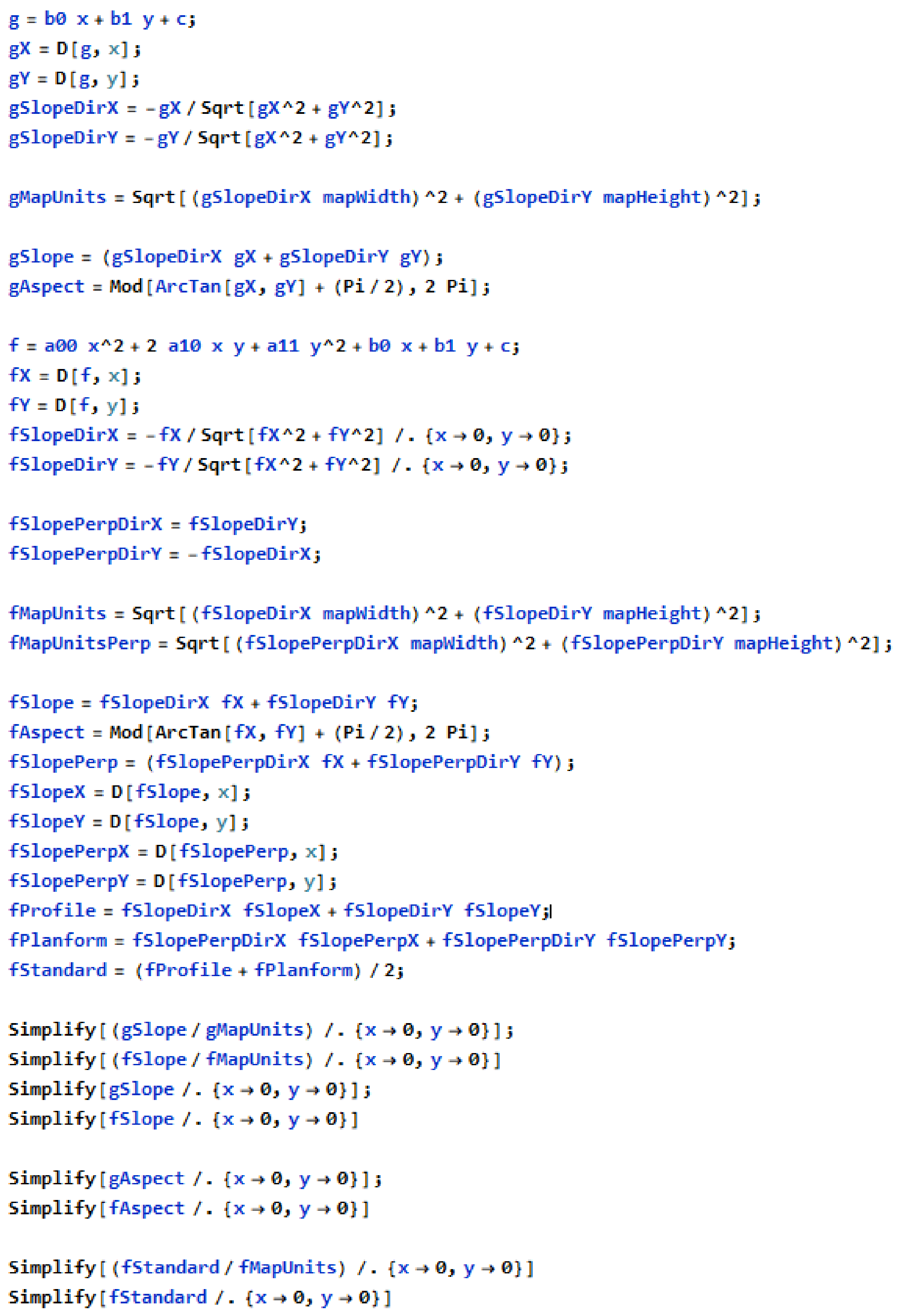
- git
  - *Change directory*
    * Windows

      $ cd \path\to\desired\directory
    * Linux

      $ cd /path/to/desired/directory
  - $ git clone git@github.com:amdenton/SlidingWindows.git
  - $ cd SlidingWindows
  - Optionally activate virtual environment, see Section 2.5.1
  - Install Rasterio, see Section 2.5.2
  - *Install requirements*
    * Windows

      $ python -m pip install -r requirements.txt
    * Linux

      $ python3 -m pip install -r requirements.txt



```
g = b0 x + b1 y + c;
gX = D[g, x];
gY = D[g, y];
gSlopeDirX = -gX / Sqrt[gX^2 + gY^2];
gSlopeDirY = -gY / Sqrt[gX^2 + gY^2];

gMapUnits = Sqrt[(gSlopeDirX mapWidth)^2 + (gSlopeDirY mapHeight)^2];

gSlope = (gSlopeDirX gX + gSlopeDirY gY);
gAspect = Mod[ArcTan[gX, gY] + (Pi / 2), 2 Pi];

f = a00 x^2 + 2 a10 x y + a11 y^2 + b0 x + b1 y + c;
fX = D[f, x];
fY = D[f, y];
fSlopeDirX = -fX / Sqrt[fX^2 + fY^2] /. {x → 0, y → 0};
fSlopeDirY = -fY / Sqrt[fX^2 + fY^2] /. {x → 0, y → 0};

fSlopePerpDirX = fSlopeDirY;
fSlopePerpDirY = -fSlopeDirX;

fMapUnits = Sqrt[(fSlopeDirX mapWidth)^2 + (fSlopeDirY mapHeight)^2];
fMapUnitsPerp = Sqrt[(fSlopePerpDirX mapWidth)^2 + (fSlopePerpDirY mapHeight)^2];

fSlope = fSlopeDirX fX + fSlopeDirY fY;
fAspect = Mod[ArcTan[fX, fY] + (Pi / 2), 2 Pi];
fSlopePerp = (fSlopePerpDirX fX + fSlopePerpDirY fY);
fSlopeX = D[fSlope, x];
fSlopeY = D[fSlope, y];
fSlopePerpX = D[fSlopePerp, x];
fSlopePerpY = D[fSlopePerp, y];
fProfile = fSlopeDirX fSlopeX + fSlopeDirY fSlopeY;
fPlanform = fSlopePerpDirX fSlopePerpX + fSlopePerpDirY fSlopePerpY;
fStandard = (fProfile + fPlanform) / 2;

Simplify[(gSlope / gMapUnits) /. {x → 0, y → 0}];
Simplify[(fSlope / fMapUnits) /. {x → 0, y → 0}]
Simplify[gSlope /. {x → 0, y → 0}];
Simplify[fSlope /. {x → 0, y → 0}]

Simplify[gAspect /. {x → 0, y → 0}];
Simplify[fAspect /. {x → 0, y → 0}]

Simplify[(fStandard / fMapUnits) /. {x → 0, y → 0}]
Simplify[fStandard /. {x → 0, y → 0}]
```

Figure 2.3. Generic analytical results generation for our linear and quadratic equations. These are mathematical derivations written in the Wolfram Mathematica programming language. The linear equation is g and the quadratic equations is f. Various calculus and algebraic operations are performed to get results for slope, aspect, and curvature. Both with map unit normalization and without.

3. DEM TESTING AND EVALUATION

To test our DEM function we will be using a double-variate Gaussian distribution. It is a very common statistical function we can use to test our library. The function itself can be seen in Equation 3.1. To get the more typical 2D variant of the Gaussian function, simply remove the expression $(-\mu + y)^2$. We can see the Gaussian 3D Wolfram plot in Figure 3.5, and its Python generated variant in Figure 3.4.

3.1. Python Image Generator

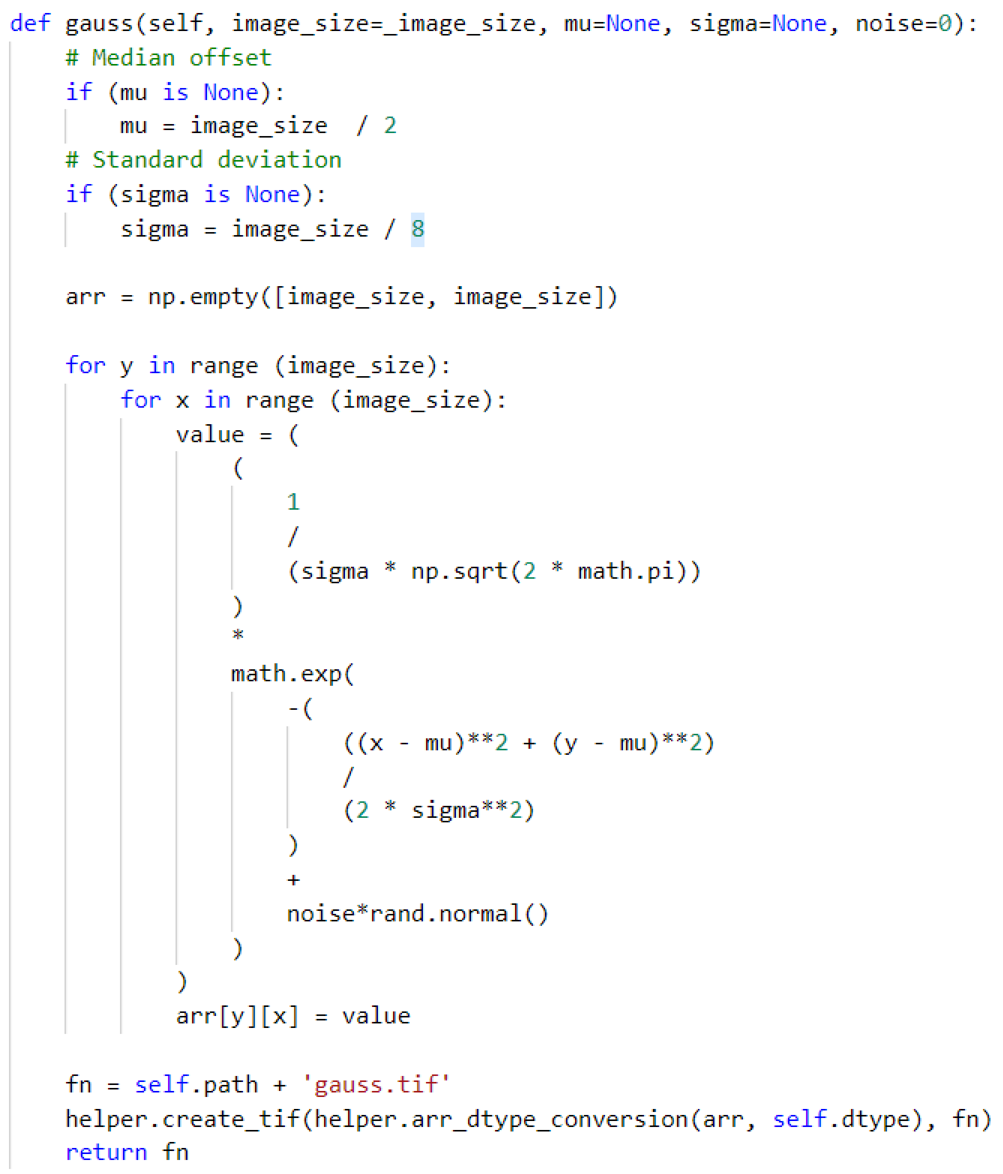
Our image generation for the Gaussian hill can be see in Figure 3.1 written in Python. It accepts an image size, offset (μ), standard deviation (σ), and noise. If the image size is not specified, a constant is used, defined at the top of the file. The μ variable is used to change the center of the Gaussian hill. Without an offset, a Gaussian hill will be centered on the origin. By default, we shift the Gaussian hill by one half of our image size, to be centered in our image. The σ variable is used to change the distribution of our Gaussian hill. Within one σ of the mean exists 68% of the volume, within two σ of the mean exists 95% of the volume, then 99.7% and so on. By default, σ is $\frac{1}{8}^{th}$ of the image size, as it is here. Lastly we have the noise, which adds slight variations to the output. It makes a slightly imperfect guassian hill.

Our image generator simply loops though both the x and y indices, then uses our Gaussian hill function with the appropriate values for x, y, μ, and σ. Our Gaussian image is saved with the helper.py file and the name of the file is returned to the caller of the function. The equation in the function is spread over multiple lines to aid in readability.

The results of our Python function can be seen in Figure 3.4. As expected, the Gaussian hill is centered in the image, and the majority of the hill exists within $\frac{1}{4}$ of the image. The reason such a small standard deviation was chosen was to enable more room for the results of the slope and curvature calculations. These will have some more interesting results than just the base Gauss function, and will cover more area.

3.2. Wolfram Image Generation

To get 3D theoretical results of what our library should be outputting, we can use Wolfram Mathematica. We can use the same process outlined in Figure 2.3 to generate our theoretical



```python
def gauss(self, image_size=_image_size, mu=None, sigma=None, noise=0):
    # Median offset
    if (mu is None):
        mu = image_size  / 2
    # Standard deviation
    if (sigma is None):
        sigma = image_size / 8

    arr = np.empty([image_size, image_size])

    for y in range (image_size):
        for x in range (image_size):
            value = (
                (
                    1
                    /
                    (sigma * np.sqrt(2 * math.pi))
                )
                *
                math.exp(
                    -(
                        ((x - mu)**2 + (y - mu)**2)
                        /
                        (2 * sigma**2)
                    )
                    +
                    noise*rand.normal()
                )
            )
            arr[y][x] = value

    fn = self.path + 'gauss.tif'
    helper.create_tif(helper.arr_dtype_conversion(arr, self.dtype), fn)
    return fn
```

Figure 3.1. A snippet from the image_generator.py file in the library. This file has many different methods designed to aid in testing the class. This method in particular is very useful for DEM testing. A multivariate Gaussian hill is generated and exported to a file on disk, with the name of the file returned to the user.

results. We will simply replace the f and g functions with our Gauss equation, as seen in Equation 3.1. We will also remove the map units, because all of our data will be synthetic. What results is Figure 3.2.

We can view the results from our equations by executing the code in Figure 3.3. Plotting the equations is done with the *Plot3D* function. A few options are passed to format our results. First we pass in our mathematical function, then we specify the range of values to evaluate over.

We are using a predefined variale named *size*, which is 4096 in our results. To ensure all of the z values are plotted we use *PlotRange -> ALL*. To label our axes we have *AxesLabel -> Automatic*. Lastly, to make our axes correspond with what we see in Python, we must reverse the direction of the y axis with *ScalingFunctions -> {None, "Reverse", None}*. This is due to the fact that $+y$ is considered downwards in raster images.

To export these images we use the *Export* function. Accepting a file name and a plotted object, we can save our plot to disk as an image. The results of the process on the Gauss equation can be seen in Figure 3.5.

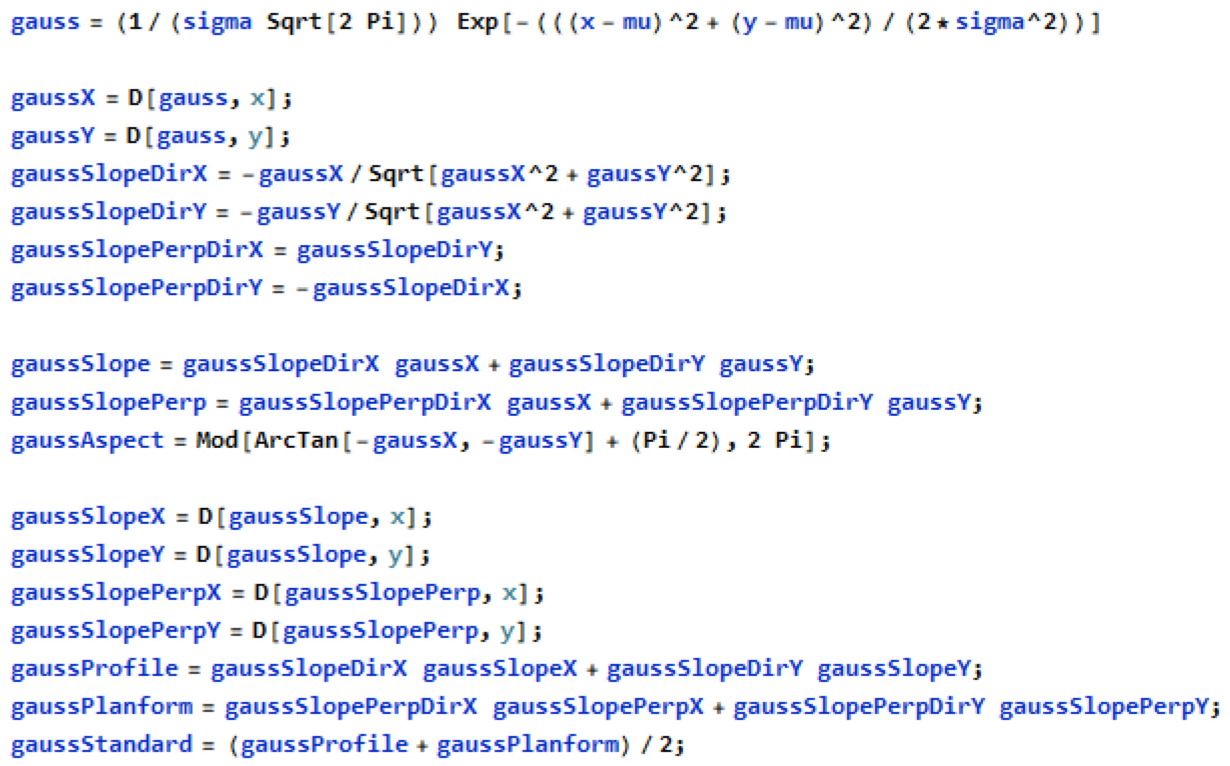
All of the following images have been confirmed to be correct within our test.py file using unit tests. First the sliding window algorithm results were generated and each pixel was compared directly to the results from analytical equations resulting from Figure 3.2. Analytical results were generated using the x and y indices of the image pixel, input into the corresponding equation. The same values for μ and σ are required for both the image generation and the analytical result generation.

3.3. Slope

When using the process in Figure 3.2 we get Equation 3.2 for our slope. The results from plotting this equation in Wolfram Mathematica can be seen in Figure 3.6. We can compare this image to the results from the sliding window algorithm in Figure 3.7. We can visually confirm the validity of the Wolfram plot with some observations. First, we expect the calculation to be in a direction extending radially outwards from the center, because this is the direction of steepest descent in a Gaussian hill. The slope in this direction is expected to be zero when there is a change in direction and when the Gaussian hill has a constant change in slope. There is potentially one occurance of this, a constant slope when the hill bottoms out at the edges of the image. This is exactly what we observe in both the Python and the Wolfram images. The center is very close to zero, but not exactly zero. We also expect the Gauss values to rapidly decrease on the curves of the hill and then taper off to zero. This is also visible in the images.

3.4. Aspect

Figure 3.2 returns Equation 3.3 when we are calculating the aspect. We use the modulo operator to keep the results in the range $[0, 2\pi)$. This modulo operator results in a few discontinuities in Mathematica. To fix these for our plotting purposes we will pass the *Exclusions -> None*



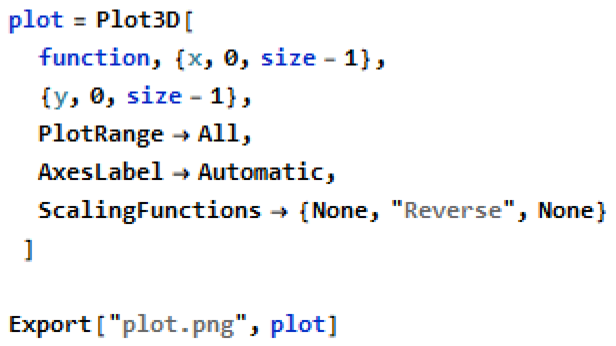
```
gauss = (1/(sigma Sqrt[2 Pi])) Exp[-(((x - mu)^2 + (y - mu)^2)/(2*sigma^2))]

gaussX = D[gauss, x];
gaussY = D[gauss, y];
gaussSlopeDirX = -gaussX/Sqrt[gaussX^2 + gaussY^2];
gaussSlopeDirY = -gaussY/Sqrt[gaussX^2 + gaussY^2];
gaussSlopePerpDirX = gaussSlopeDirY;
gaussSlopePerpDirY = -gaussSlopeDirX;

gaussSlope = gaussSlopeDirX gaussX + gaussSlopeDirY gaussY;
gaussSlopePerp = gaussSlopePerpDirX gaussX + gaussSlopePerpDirY gaussY;
gaussAspect = Mod[ArcTan[-gaussX, -gaussY] + (Pi/2), 2 Pi];

gaussSlopeX = D[gaussSlope, x];
gaussSlopeY = D[gaussSlope, y];
gaussSlopePerpX = D[gaussSlopePerp, x];
gaussSlopePerpY = D[gaussSlopePerp, y];
gaussProfile = gaussSlopeDirX gaussSlopeX + gaussSlopeDirY gaussSlopeY;
gaussPlanform = gaussSlopePerpDirX gaussSlopePerpX + gaussSlopePerpDirY gaussSlopePerpY;
gaussStandard = (gaussProfile + gaussPlanform)/2;
```

Figure 3.2. Gaussian analytical result generation written in Wolfram Mathematica. This follows the same process as in Figure 2.3, except with a multivariate Gaussian function. Results are computed for slope, aspect, and curvature. Results are saved to variables such as *gaussSlope* and *gaussStandard*, but are not printed to the user.



```
plot = Plot3D[
  function, {x, 0, size - 1},
  {y, 0, size - 1},
  PlotRange -> All,
  AxesLabel -> Automatic,
  ScalingFunctions -> {None, "Reverse", None}
 ]

Export["plot.png", plot]
```

Figure 3.3. 3D plot generation of analytical results written in Wolfram Mathematica. Plot3D function is used for generation and Export function is used for saving the image. The word ”plot” is used as a placeholder for any double variate mathematical function.

option to the *Plot3D* function. Our plotted Wolfram Mathematica results are in Figure 3.8. We can compare these to our sliding window algorithm results in Figure 3.9.

Visualizing the apsect results is fairly straightforward. We are still doing calculations radially from the center at the point $(2048, 2048)$. There are no flat faces on the hill of a Gaussian surface, we expect the aspect to be unique in every radial direction from the center. Calculating from due North, we expect values in North-Eastern directions to be the lowest, and values in North-Western directions to be the highest. This is what we see in our results from Wolfram and Python, although the wolfram results seem to be slightly more curved than the perfect corkskrew shape we expect. This is potentially a result of the discontinuities from our modulus operation. In addition, we see black in the four corners of our Python image. This can potentially be explained by a lack of data due to having a zero slope in those areas.

3.5. Curvature

Our last computation is curvature. Computed again from Figure 3.2, we see the analytical results in Equation 3.4, Wolfram results in Figure 3.10, and sliding window algorithm results can be seen in Figure 3.11. To visually confirm these images, we must look at the slope images, the Python results in Figure 3.7 are helpful in particular.

We will first consider the curvature in the direction of steepest descent, the profile curvature. Observing movement radially from the center, in the Python image we can see that the slope is relatively constant in a black band around the middle of the hill and along the edges of the image. We can also see that the slope starts with a large negative change in value and then quickly rises to a positive value and ends at zero.

Now we will consider the curvature perpendicular to the steepest descent, the planform curvature. We can imagine that since the Gaussian hill is a perfect circle, with every direction of steepest descent extending radially from the center, movement perpendicular to this would be zero.

Averaging the results of the profile and planform curvature, we can ignore the planform results because they are zero. Considering the theoretical profile results, we see the spots in the curvature where we expected zero results. We also see locations where we expected the slope to first be negative and then increase, leveling off to zero near the edges of the image.

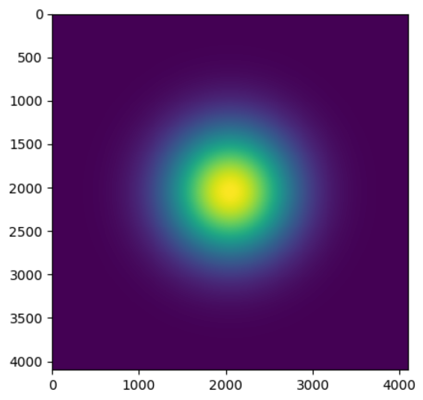


Figure 3.4. Image generated by the Gauss function in the image_generater.py file. An image size of 4096 pixels square was used with a standard deviation (σ) of 512 pixels and an offset (μ) of 2048 pixels.

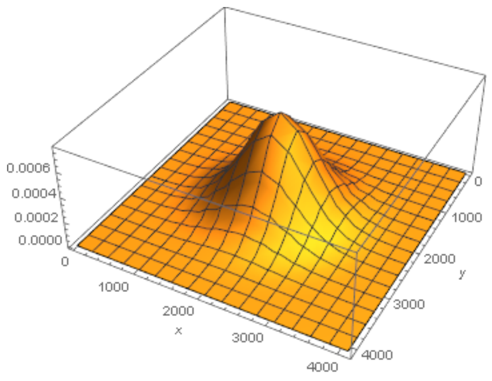


Figure 3.5. Results from using the Wolfram Mathematica Plot3D function on the multivariate Gauss function. An image size of 4096 pixels square was used with a standard deviation (σ) of 512 pixels and an offset (μ) of 2048 pixels

$$\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(-\mu+x)^2+(-\mu+y)^2}{2\sigma^2}} \tag{3.1}$$

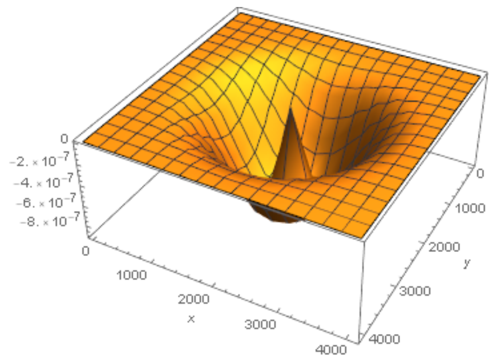


Figure 3.6. Results from using the Wolfram Mathematica Plot3D function on the analytical slope results of the Gauss function. An image size of 4096 pixels square was used with a standard deviation (σ) of 512 pixels and an offset (μ) of 2048 pixels

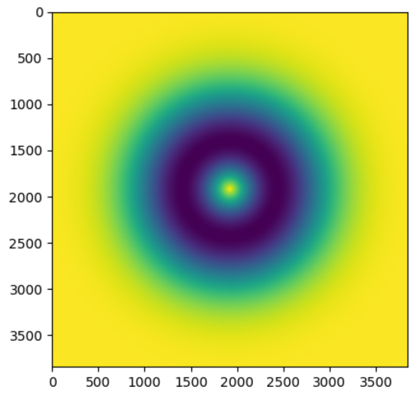


Figure 3.7. Image generated when the slope function of the sliding window library operated on a Gaussian hill. The Gaussian hill had an image size of 4096 pixels square with a standard deviation (σ) of 512 pixels and an offset (μ) of 2048 pixels. Eight aggregation steps were used for a total sliding window size of 256 pixels.

$$-\frac{\sqrt{\frac{e^{-\frac{(\mu-x)^2+(\mu-y)^2}{\sigma^2}}(2\mu^2+x^2+y^2-2\mu(x+y))}{\sigma^6}}}{\sqrt{2\pi}} \tag{3.2}$$

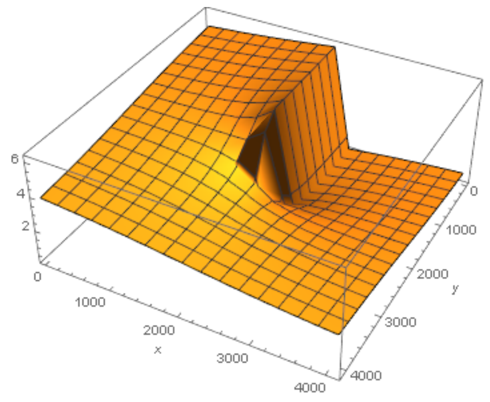


Figure 3.8. Results from using the Wolfram Mathematica Plot3D function on the analytical aspect results of the Gauss function. The *Exclusions -> None* option was used to remove discontinuities. An image size of 4096 pixels square was used with a standard deviation (σ) of 512 pixels and an offset (μ) of 2048 pixels

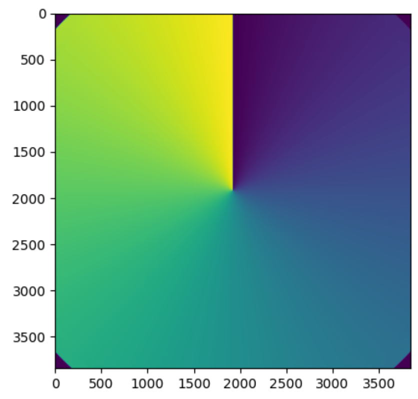


Figure 3.9. Image generated when the aspect function of the sliding window library operated on a Gaussian hill. The Gaussian hill had an image size of 4096 pixels square with a standard deviation (σ) of 512 pixels and an offset (μ) of 2048 pixels. Eight aggregation steps were used for a total sliding window size of 256 pixels.

$$\left(\operatorname{atan2}\left(\frac{e^{-\frac{(\mu-x)^2+(\mu-y)^2}{2\sigma^2}}(-\mu+x)}{\sigma^3\sqrt{2\pi}}, \frac{e^{-\frac{(\mu-x)^2+(\mu-y)^2}{2\sigma^2}}(-\mu+y)}{\sigma^3\sqrt{2\pi}}\right)+\frac{\pi}{2}\right) \mod 2\pi \tag{3.3}$$

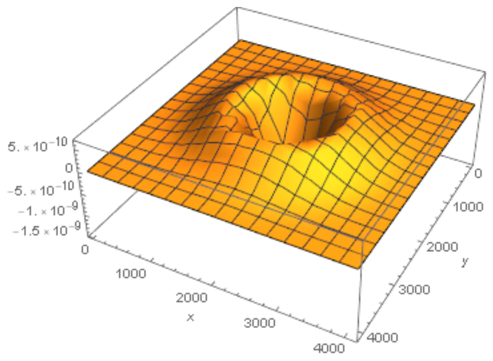


Figure 3.10. Results from using the Wolfram Mathematica Plot3D function on the analytical standard curvature results of the Gauss function. An image size of 4096 pixels square was used with a standard deviation (σ) of 512 pixels and an offset (μ) of 2048 pixels

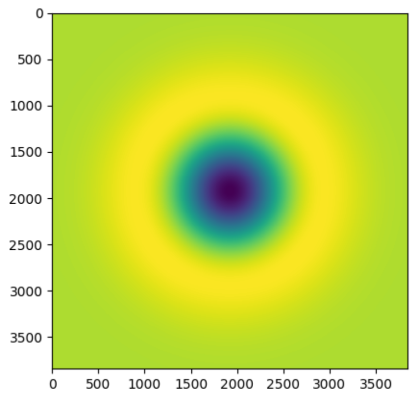


Figure 3.11. Image generated when the standard function of the sliding window library operated on a Gaussian hill. The Gaussian hill had an image size of 4096 pixels square with a standard deviation (σ) of 512 pixels and an offset (μ) of 2048 pixels. Eight aggregation steps were used for a total sliding window size of 256 pixels.

$$\frac{e^{-\frac{2\mu^2+x^2+y^2-2\mu(x+y)}{2\sigma^2}}\left(2\mu^2-\sigma^2+x^2+y^2-2\mu(x+y)\right)}{2\sigma^5\sqrt{2\pi}} \tag{3.4}$$

4. RESULTS

Not only was this library created to easily switch between analyses without switching software packages, but also to combine these analyses into a single image. Code for these types of operations exist in the previously mentioned cluster.py file. In this section we will cover a few example uses of this file.

4.1. Spectral

Our first example is an analysis of a National Agricultural Imagery Program (NAIP) image taken on the border of North and South Dakota. It is a hyperspectral RBG/IR image, so we will be analyzing it using our rbg.py file. In this scenario we are going to do a spectral correlation between the red and green bands. We want to see how this result correlates with itself when calculated with varying sliding window sizes. We will arbitrarily be calculating this analysis with the number of aggregations varying in the range $[3, 7]$. That is, a sliding window size varying in the range $[8, 128]$, in powers of two. We will make our sub image 512 pixels long, 4 times the size of our largest aggregation, to allow for some breathing room. To make this calculation we will pass the following to our function: *analyses=[Analysis.regression, Analysis.regression, Analysis.regression, Analysis.regression, Analysis.regression]*, *bands=[[1, 2], [1, 2], [1, 2], [1, 2], [1, 2]]* for our red and green bands, *num_aggres=[3, 4, 5, 6, 7]*, and *sub_img_size=512*.

Our adjusted image can be seen in Figure 4.1. This image is a 384x384 pixel image taken from the very top left corner of the original. It was a 512x512 pixel image, which had $\frac{2^7-1}{2} = 63.5$ pixels removed from each side, to match our analyses output which has been aggregated 7 times.

The results of our pair plot can be seen in Figure 4.2. We can see in this image that all of the window sizes correlate fairly strongly having a regression slope that is approximately one. The correlation is the strongest when aggregation windows are a factor of two away from each other. For example, the correlation between the window sizes of 8 and 16 is very strong. This makes sense when we observe that the histograms of all the analyses on the main diagonal have the same general shape. The histograms get more varied as the window size increases due to the increase in data. It is also interesting to note that the window size of 128 froms a hook shape when correlated with other window sizes. Values of .5 evidently tend to vary more as the size of the window increases to 128 pixels.

4.2. Digital Elevation Model

Our second example will be a DEM file take in Arizona. We will analyze this file using the dem.py file. We will be trying to figure out how all of the DEM functions and fractal functions are related. This will be done with both the *gen_clustered_img* and the *gen_pairplot_img* functions. Again, we will be varying the number of aggregations in the range $[3, 7]$. We will make our sub image 1000 pixels long at position $(1000, 1000)$ because we have a very high resolution image and want to capture a whole mountain feature in our image. To make this calculation we will pass the following to our function: *analyses=[Analysis.slope, Analysis.aspect, Analysis.standard, Analysis.fractal, Analysis.fractal_3d]*, *bands=[1, 1, 1, 1, 1]*, *num_aggres=[3, 4, 5, 6, 7]*, *sub_img_start=[1000, 1000]*, and *sub_img_size=512*. The adjusted image is in Figure 4.3.

Results of the *gen_clustered_img* function can be seen in Figure 4.5. We have used the argument *average_cluster_values=True* to visualize the general magnitude of the cluster centers. The most obvious thing to notice is that the aspect has a very large effect on the clustering. Parts of the mountain facing in a general South-Western direction are darker and regions facing generally in a North-Eastern direction have have lighter colors. We can also see that colors near the peak of the mountain have higher values, presumably because the slope and curvature are higher there. Lastly, we could also explain the darker colors in the lower left of the image by the fractal dimensions. There are generally rougher structures in the lower left corner, which would correlate with a higher fractal dimension.

Our results for the *gen_pairplot_img* function can be seen in Figure 4.4. The most obvious correlation is between the slope and the fractal_3d analyses. A higher slope seems to correlate with a rougher 3D shape. Observing the 2D fractal dimensions we see the values seems to be equally split between the integer dimensions of 0 and 2. This means most features are either points or planes. Planes seem to have lower 3D fractal dimensions, probably containing less changes in elevation. This is also reflected in the slope, with lower slope slightly correlating with planes from the 2D fractal dimensions. The aspect appears to be equally dispersed among the other analyses, with generally vertical and horizontal lines for correlation. Lastly, the curvature doesn't seem to correlate with anything at all, logically a very separate analysis.

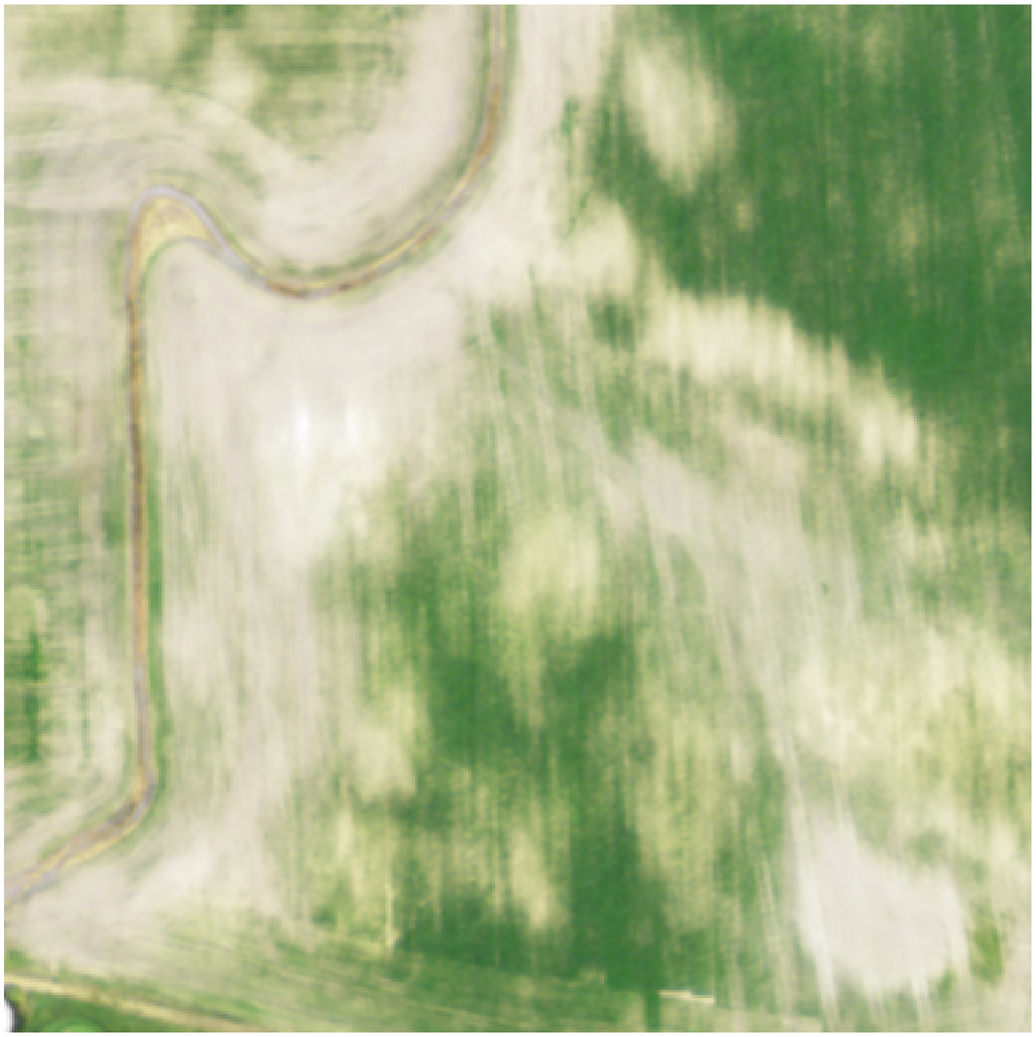


Figure 4.1. The adjusted image from a pair plot operation on a National Agricultural Imagery Program (NAIP) file. Resulting from the maximum number of aggregations equaling 7, *sum_img_size = 512*, and *sub_img_start = [0, 0]*.

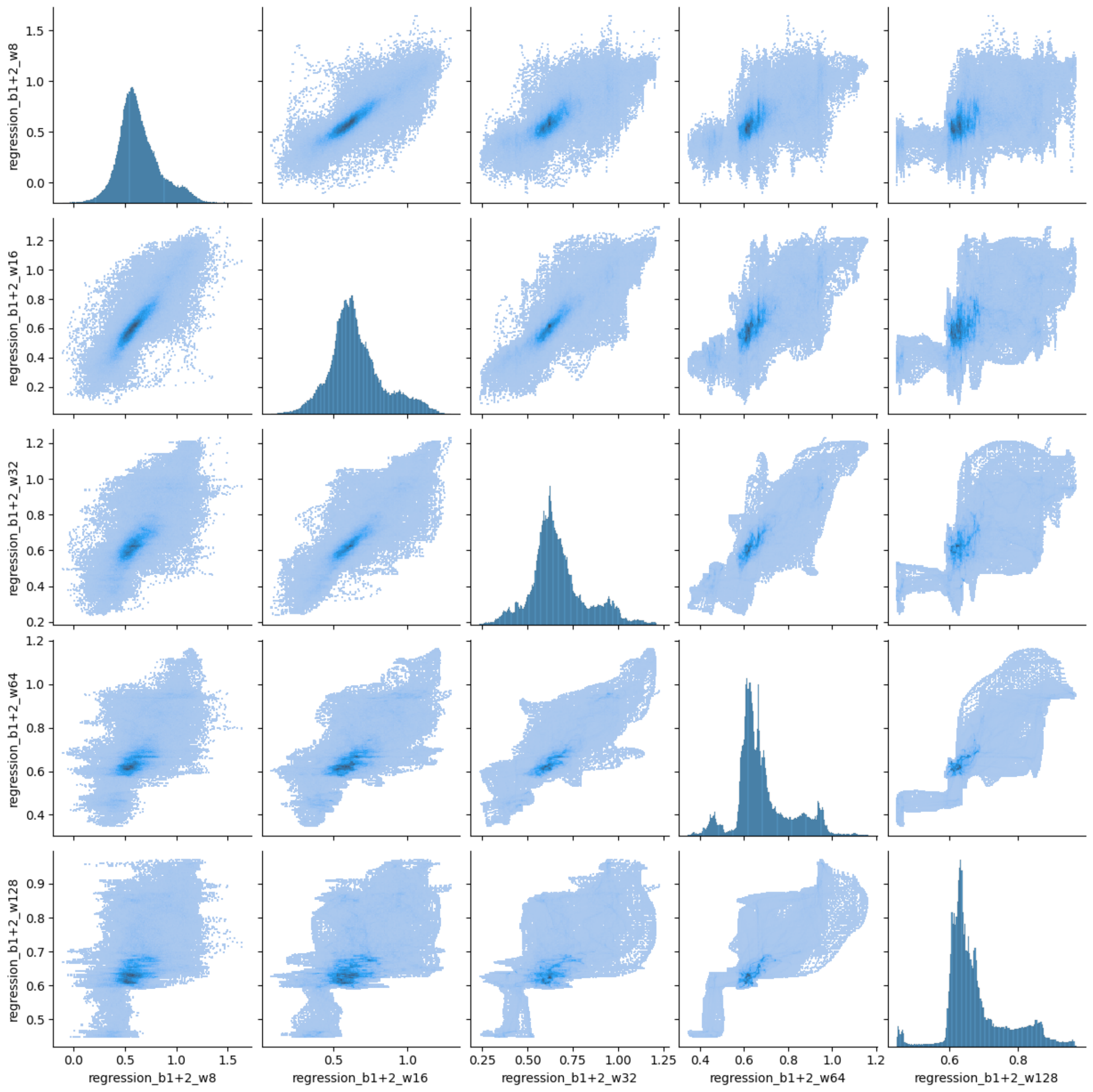


Figure 4.2. The output image from a pair plot operation on a National Agricultural Imagery Program (NAIP) file. Calculated with the following parameters: *sum_img_size = 512*, *sub_img_start = [0, 0]*, *analyses = [Analysis.regression, Analysis.regression, Analysis.regression, Analysis.regression, Analysis.regression]*, *bands = [[1, 2], [1, 2], [1, 2], [1, 2], [1, 2]]*, and *num_aggres = [3, 4, 5, 6, 7]*.

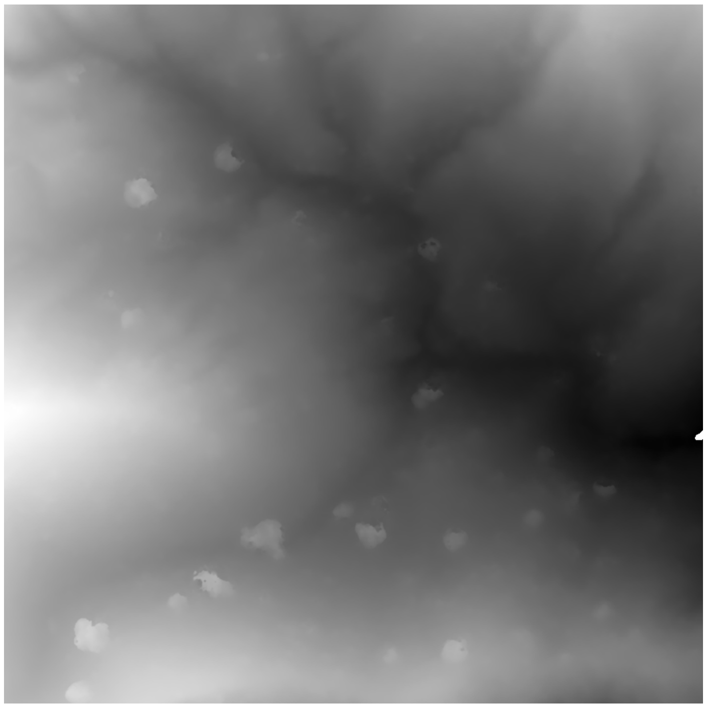


Figure 4.3. The adjusted image from a pair plot operation on a DEM file. Resulting from the maximum number of aggreagations equaling 7, *sum_img_size = 1000*, and *sub_img_start = [1000, 1000]*.

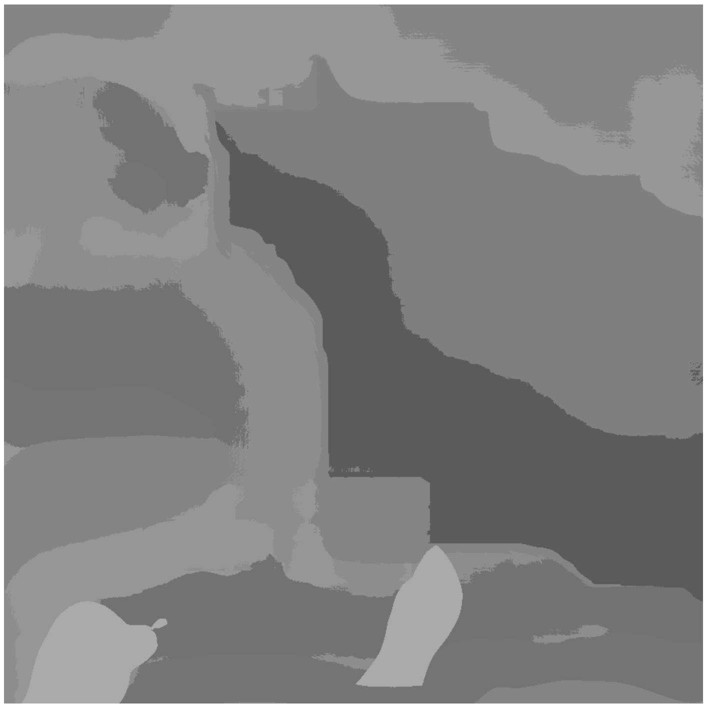


Figure 4.4. The output image from a cluster operation on a DEM file. Calculated with the following parameters: *sum_img_size = 1000*, *sub_img_start = [1000, 1000]*, *num_clusters = 10*, *average_cluster_values = True*, *analyses = [Analysis.slope, Analysis.aspect, Analysis.standard, Analysis.fractal, Analysis.fractal_3d]*, *bands = [1, 1, 1, 1, 1]*, and *num_aggres = [7, 7, 7, 7, 7]*.

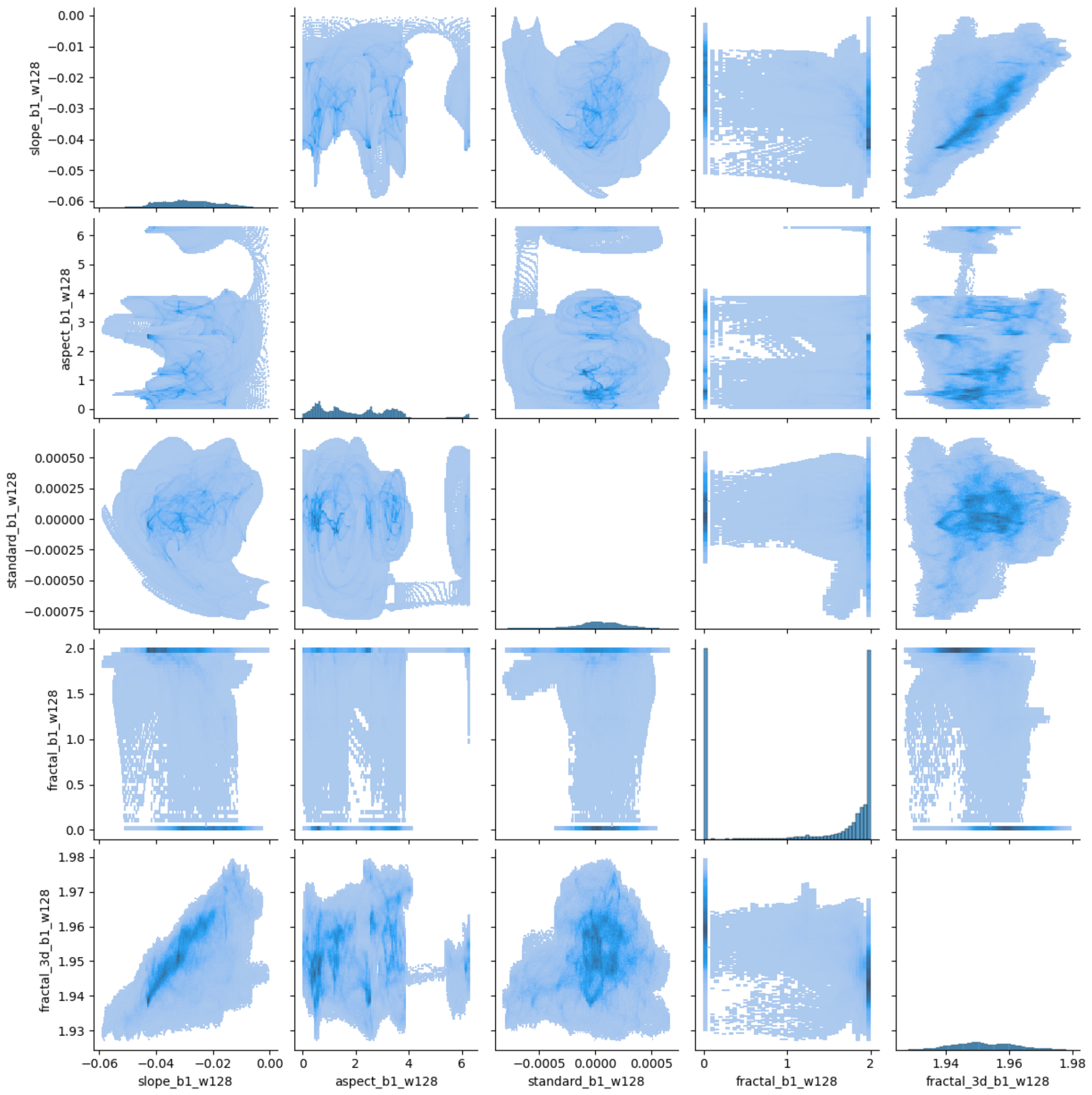


Figure 4.5. The output image from a pair plot operation on a DEM file. Calculated with the following parameters: *sum_img_size = 1000*, *sub_img_start = [1000, 1000]*, *analyses = [Analysis.slope, Analysis.aspect, Analysis.standard, Analysis.fractal, Analysis.fractal_3d]*, *bands = [1, 1, 1, 1, 1]*, and *num_aggres = [7, 7, 7, 7, 7]*.

5. CONCLUSION

GIS methods can be improved to better handle topographical calculations. A sliding window algorithm can be utilized to create a library to execute a variety of analyses. These analyses are efficient, due to the nature of the algorithm's $O(\log(n))$ time complexity, and can be combined in interesting ways.

We have developed a Python library. Enabling the code readability of Python while also having the performance of C code using a package called NumpPy. This library was designed to be open source and have a high correlation with theoretical results to enable high comprehension, readability, and modularity of code. Methods within the library were separated into files of varying levels of abstraction, using the design principle of data abstraction. Code was separated into groups relating to either aggregations, DEM files, spectral images, or analysis synthesis.

Validation of this library was performed using brute-force techniques of $O(n^2)$ time complexity and analytical results derived in Wolfram Mathematica. Through direct comparison of analytical results & brute-force techniques to the results of the sliding window algorithm, validation was acheived. Visualizations of 3D Wolfram Mathematica results were generated using the built in *Plot3D* function. Visualizations were compared intuitively to the sliding window algorithm's DEM outputs. Using calculus intuition, we saw it was reasonble to assume, even without our unit test results, that our results were correct.

To tie our library all together, we outlined two operations enabling the synthesis of many differnt analyses. With either pair plots or k-means clustering, we combined our analyses. Using our pair plots we saw surprising ways in which different analyses were correlated. Using clustering, we saw the regions of an image where different analyses could be related with a common cluster group. We understood that additional statistical methods could be added to this library in the future through open source innovation.

REFERENCES

[1] John W Allan. High resolution geographic imagery and its impact on gis, Jun 2017.

[2] Dayna Behm, Tony Bryan, Joshua Lordemann, and Steven R Thomas. The past, present, and future of geospatial data use, Feb 2018.

[3] Laura M Castro. It was never about the language: paradigm impact on software design decisions. 2020.

[4] Caitlin Dempsey. What is gis?, Oct 2020.

[5] A. M. Denton, M. Ahsan, D. Franzen, and J. Nowatzki. Multi-scalar analysis of geospatial agricultural data for sustainability. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2139–2146, 2016.

[6] Anne Denton, Rahul Gomes, and David Franzen. Scaling up window-based slope computations for geographic information system. In *2018 IEEE International Conference on Electro/Information Technology (EIT)*, pages 0554–0559. IEEE, 2018.

[7] Michael D Fleming and Roger M Hoffer. Machine processing of landsat mss data and dma topographic data for forest cover type mapping. In *LARS Symposia*, page 302, 1979.

[8] J.J. Gagnepain and C. Roques-Carmes. Fractal approach to two-dimensional and three-dimensional surface roughness. *Wear*, 109(1):119–126, 1986.

[9] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández Del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E Oliphant. Array programming with numpy. *Nature (London)*, 585(7825):357–362, 2020.

[10] Kevin H Jones. A comparison of two approaches to ranking algorithms used to compute hill slopes. *GeoInformatica*, 2(3):235–256, 1998.

[11] Herbert J. Kramer and Arthur P. Cracknell. An overview of small satellites in remote sensing. *International Journal of Remote Sensing*, 29(15):4285–4337, 2008.

[12] Benoit Mandelbrot. How long is the coast of britain? statistical self-similarity and fractional dimension. *science*, 156(3775):636–638, 1967.

[13] NASA. Landsat 7.

[14] Trisalyn A. Nelson. Trends in spatial statistics. *The Professional Geographer*, 64(1):83–94, 2012.

[15] David A Sharpnack. An algorithm for computing slope and aspect from elevations. *Photogrammetric Engineering*, 35(3):247–248, 1969.

[16] B. Stroustrup. What is object-oriented programming? *IEEE Software*, 5(3):10–20, 1988.

[17] U.S. Geological Survey. Landsat missions.

[18] U.S. Geological Survey. What are the band designations for the landsat satellites?

[19] Huang Yao, Rongjun Qin, and Xiaoyu Chen. Unmanned aerial vehicle for remote sensing applications—a review. *Remote Sensing*, 11:1443, 06 2019.