Terrain Modelling with Generative AI

Authors: Ewan Walker, Harry Fletcher, Ed Warneken
Supervisor: Dr Nick Keepfer
MAS3091 2023-2024 – Group 8

Newcastle University

Overview

A detailed outline of the objectives are as follows:

1. Code multiple of these methods for analysis by varying parameters and create plots, colourmaps and 3D projections (where applicable). Implement reproducibility via the use of seeds (an initialisation of pseudorandom number generators to recreate the same 'random' terrain). Utilise noise in heightmaps – fluently blended 'random' normalised values (ranging from 0 to 1) sorted in matrices to create 3D projections using the values as height components.
2. Explore principal component analysis to identify the most important data-points representing the terrain and eliminate any excess for dimensionality reduction, thereby compressing the volume of data. Find out the optimum number of principle components to minimize loss of quality when compared to the original terrain.
3. Build autoencoders, trained on the generated heightmaps, to create a 2D grid in which clusters of points correlate to specific features of the terrain. These are much more efficient than traditional methods.
4. Analyse real-world LiDAR scans via the autoencoders to create realistic, new, predictable terrain from a few select coordinates.

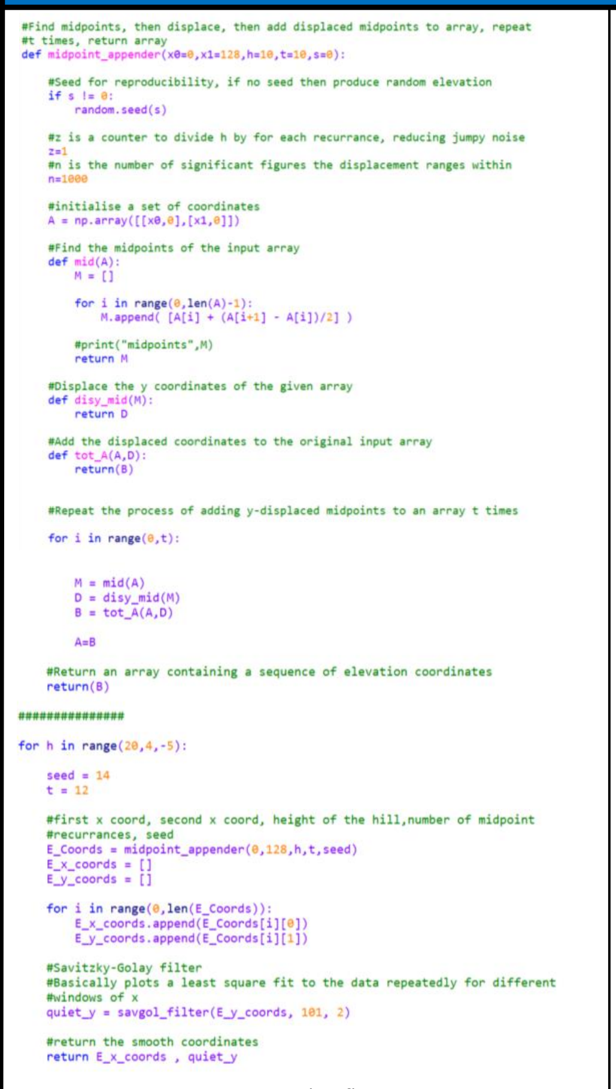
Clearly, that is a lot to unpack! However, the simplistic beauty of the process can be quickly illustrated, and all the fancy concepts and definitions are reduced to an easily understandable format.

Firstly, to understand what we mean by 'terrain', take a widely recognised example: Minecraft. In Minecraft, vast, unique worlds are near-instantaneously spawned into existence for the player to explore. They are random; they also contain consistently predictable patterns. One can reproduce infinitely generating, complex random worlds using a short string of numbers, a seed, whilst no two seeds are the same. How can this be? Below is information to lay a bedrock understanding of all these processes.

[1]Minecraft Landscape: a prime example of terrain generation

Standard Methods

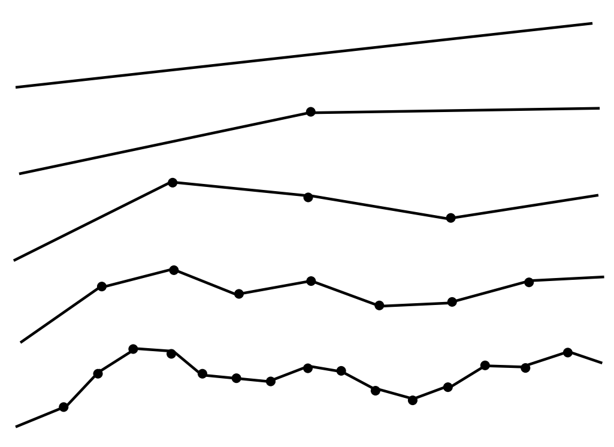
Midpoint Displacement

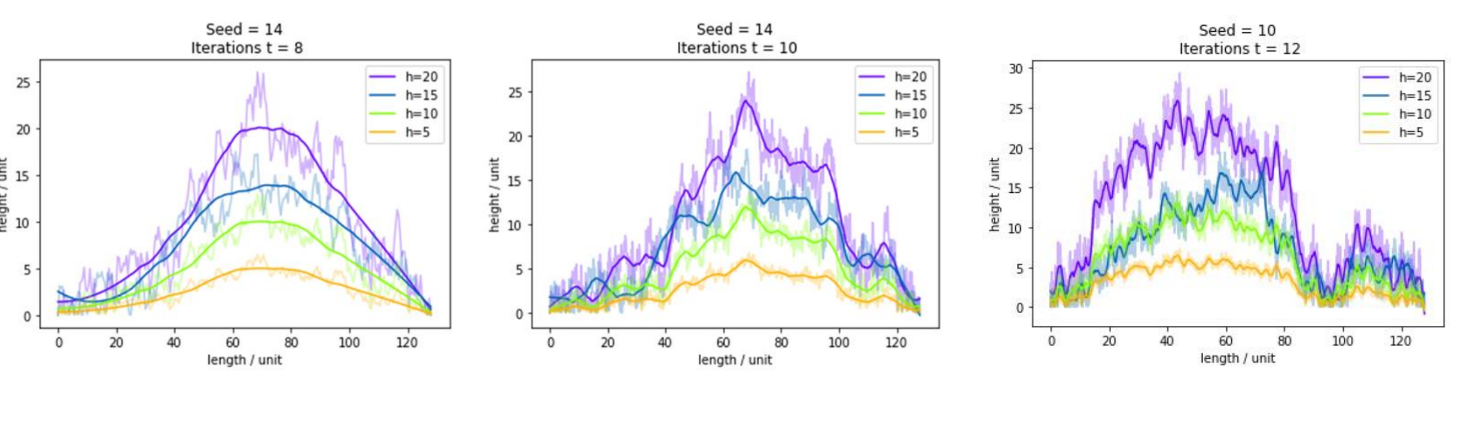


```
#Find midpoints, then displace, then add displaced midpoints to array, repeat
#t times, return array
def midpoint_appender(x0=0,x1=128,h=10,t=10,s=0):

    #Seed for reproducibility, if no seed then produce random elevation
    if s != 0:
        random.seed(s)

    #z is a counter to divide h by for each recurrance, reducing jumpy noise
    z=1
    #n is the number of significant figures the displacement ranges within
    n=1000

    #initialise a set of coordinates
    A = np.array([[x0,0],[x1,0]])

    #Find the midpoints of the input array
    def mid(A):
        M = []

        for i in range(0,len(A)-1):
            M.append( [A[i] + (A[i+1] - A[i])/2] )

        #print("midpoints",M)
        return M

    #Displace the y coordinates of the given array
    def disy_mid(M):
        return D

    #Add the displaced coordinates to the original input array
    def tot_A(A,D):
        return(B)


    #Repeat the process of adding y-displaced midpoints to an array t times

    for i in range(0,t):


        M = mid(A)
        D = disy_mid(M)
        B = tot_A(A,D)

        A=B

    #Return an array containing a sequence of elevation coordinates
    return(B)

###############

for h in range(20,4,-5):

    seed = 14
    t = 12

    #first x coord, second x coord, height of the hill,number of midpoint
    #recurrances, seed
    E_Coords = midpoint_appender(0,128,h,t,seed)
    E_x_coords = []
    E_y_coords = []

    for i in range(0,len(E_Coords)):
        E_x_coords.append(E_Coords[i][0])
        E_y_coords.append(E_Coords[i][1])

    #Savitzky-Golay filter
    #Basically plots a least square fit to the data repeatedly for different
    #windows of x
    quiet_y = savgol_filter(E_y_coords, 101, 2)

    #return the smooth coordinates
    return E_x_coords , quiet_y
```

Midpoint Displacement (half) pseudo-code

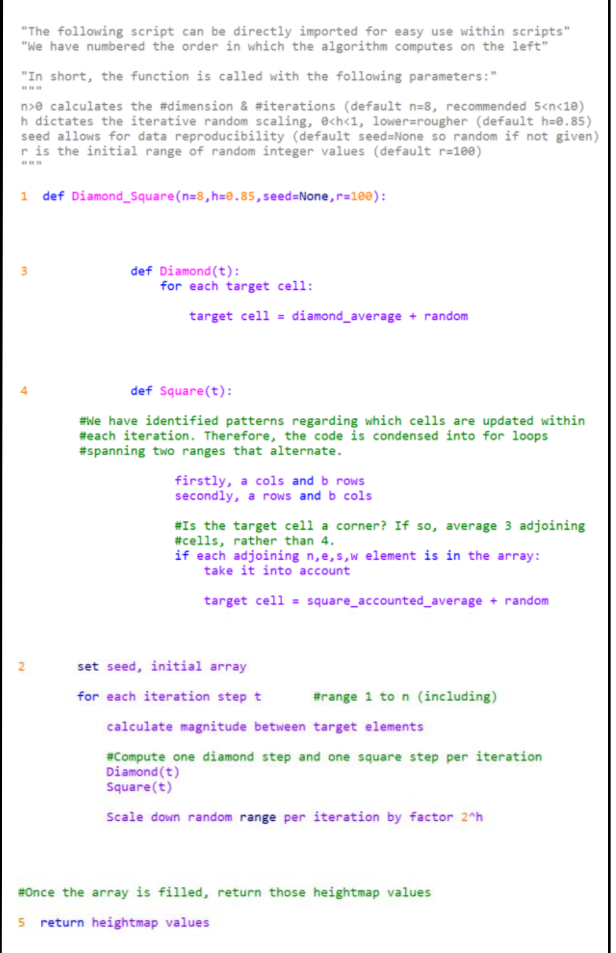
The Midpoint Displacement method begins with taking two end points of a 1D line and iteratively calculates the midpoint of the connecting line, displacing each point in the y-direction with an incrementally reduced range of randomness (scaled by a h parameter which depends on iteration number t).


Midpoint Displacement Theory visualised.



Shows the result for a seed with various scaling factor 'h' parameter. The shadows are generated using the code and the solid lines utilise the Savitzky-Golay Filter[2] (MSE line) for smoothness.

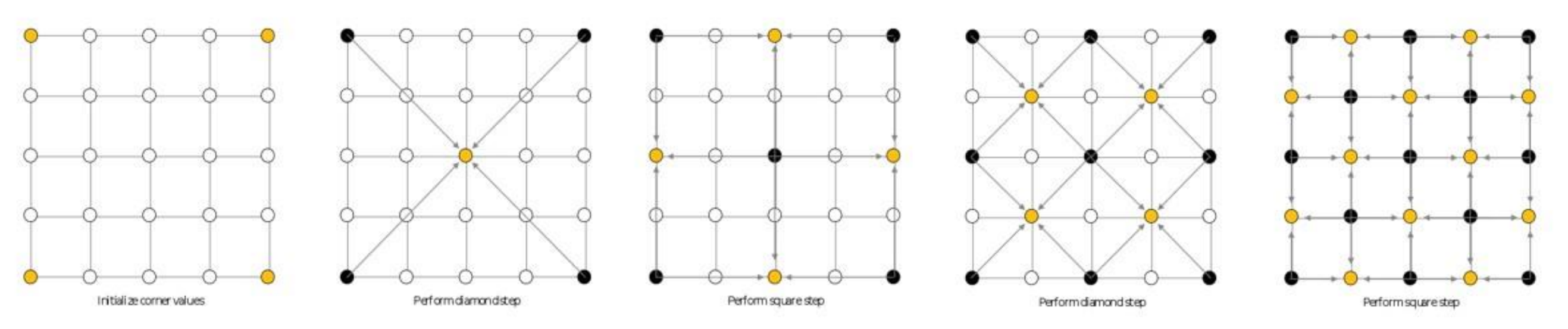
Diamond Square



```
"The following script can be directly imported for easy use within scripts"
"We have numbered the order in which the algorithm computes on the left"

"In short, the function is called with the following parameters:"
"""
n>0 calculates the #dimension & #iterations (default n=8, recommended 5<n<10)
h dictates the iterative random scaling, 0<h<1, lower=rougher (default h=0.85)
seed allows for data reproducibility (default seed=None so random if not given)
r is the initial range of random integer values (default r=100)
"""

1  def Diamond_Square(n=8,h=0.85,seed=None,r=100):



3             def Diamond(t):
                  for each target cell:

                      target cell = diamond_average + random



4             def Square(t):

          #We have identified patterns regarding which cells are updated within
          #each iteration. Therefore, the code is condensed into for loops
          #spanning two ranges that alternate.

                      firstly, a cols and b rows
                      secondly, a rows and b cols

                      #Is the target cell a corner? If so, average 3 adjoining
                      #cells, rather than 4.
                      if each adjoining n,e,s,w element is in the array:
                          take it into account

                          target cell = square_accounted_average + random


2         set seed, initial array

          for each iteration step t       #range 1 to n (including)

              calculate magnitude between target elements

              #Compute one diamond step and one square step per iteration
              Diamond(t)
              Square(t)

              Scale down random range per iteration by factor 2^h


#Once the array is filled, return those heightmap values

5  return heightmap values
```

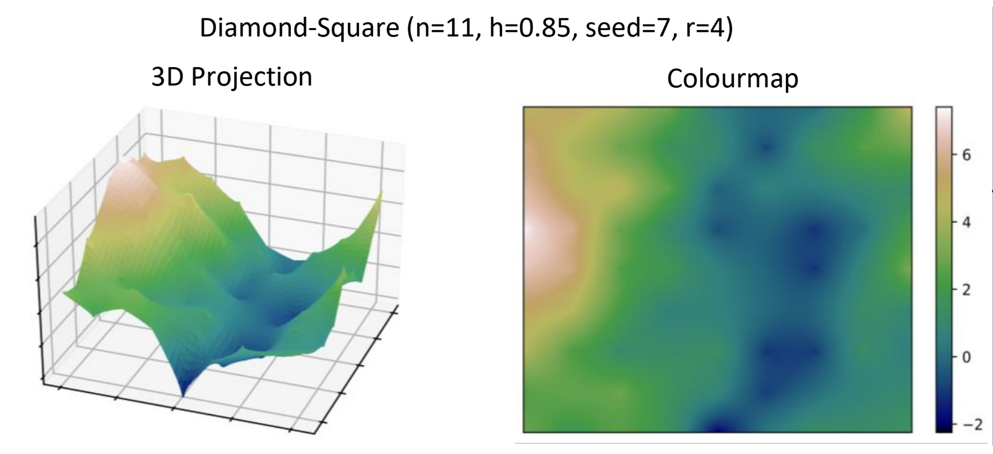
Diamond Square pseudo-code

This method begins with a 2D grid and creates a matrix of heightmap values. These essentially are the z coordinates used for generating a 3D surface. Starting with a 2D array, initialise the 4 corners with some random value. Firstly, a diamond step takes an average of the 4 nearest corners and adds some random noise - think of this as diagonal. Secondly, a square step takes an average of the nearest horizontal and vertical points (an average of 3 if an edge) with additional noise. These are repeated until the array is full, decreasing the range of randomness with each iteration.



[3]A visualisation of the Diamond-Square process.

After finding patterns in each iteration (one diamond step and one square step) we have identified formulas to make an efficient algorithm function successfully. Attached are figures of the pseudo-code and an example of how we use loops in python to target each cell, depending on the current iteration t.



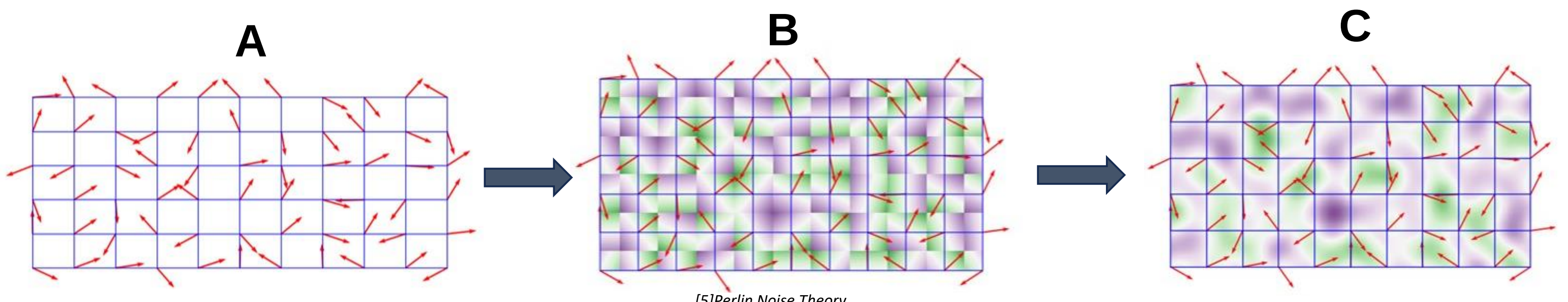
$$t = 1, 2, \dots n \qquad m_t = \frac{d-1}{t} \qquad e_t = \frac{m_t}{2} \qquad a_{D,t} = [0, 1, \dots, \sqrt{C_{D,t}}]$$

$$d = 2^n + 1 \qquad C_{D,t} = 2^{2(t-1)} \qquad b_{D,t} = [0, 1, \dots, \sqrt{C_{D,t}}]$$

$$\begin{bmatrix} [a_{D,t}m_t][b_{D,t}m_t] & & [a_{D,t}m_t][(b_{D,t}+1)m_t] \\ & [a_{D,t}m_t + e_t][b_{D,t}m_t + e_t] & \\ [(a_{D,t}+1)m_t][b_{D,t}m_t] & & [(a_{D,t}+1)m_t][(b_{D,t}+1)m_t] \end{bmatrix}$$

Perlin Noise

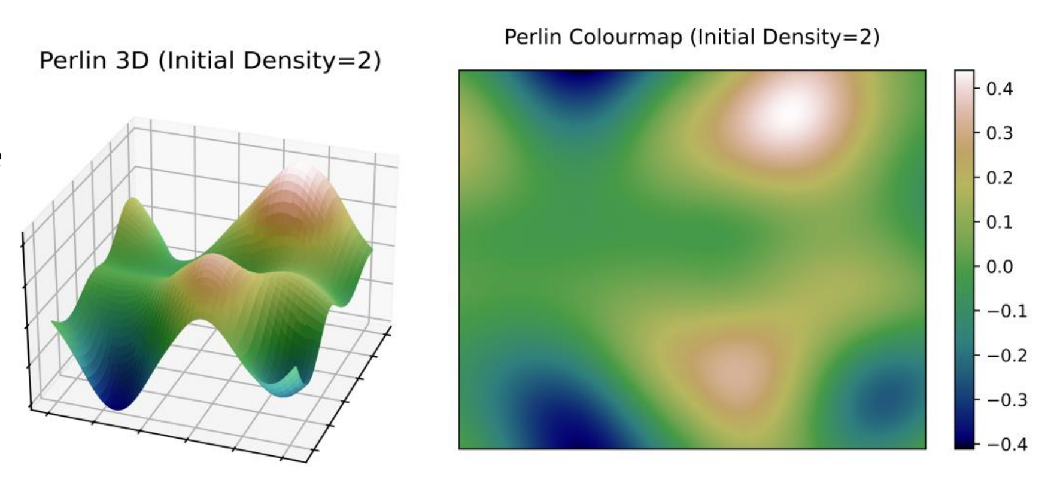
This method was developed by Ken Perlin [4] and creates a heightmap for any dimension. Here, we look at 2D to 3D. The process is simply demonstrated as follows, from A > B > C:



[5]Perlin Noise Theory.

1. Create a 2D grid of cells and assign each intersection with a unit vector of gradient between -1 and 1.
2. For every pixel in the whole grid, calculate the displacement vector (d) of said point with its nearest cell corner.
3. Calculate the dot product of all these d vectors and the corresponding corner vectors. This is shown in image B (colour coordinated).
4. Repeat this process but to the other 3 corners of each cell to get 4 total sets of gradients.
5. Combine the 4 sets of the gradients (dot product values) together using a "smoothstep function" to get to the heightmap C.
6. Numerous versions of these are then combined to increase clarity. Known as octaves, the density of the initial grid increases, these are added together with a decreased weight / transparency.

The final result produces incredible, smooth terrain.


Perlin 3D (Initial Density=2)
Perlin Colourmap (Initial Density=2)
A heighmap (coloured) and 3D projection, both generated using a Perlin Noise [6] python package.

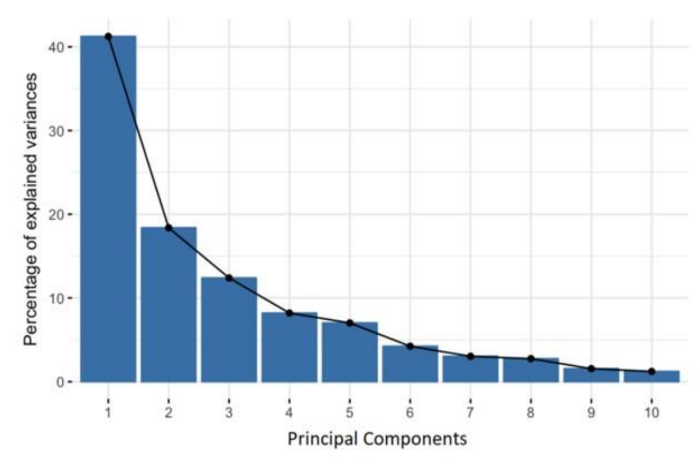
Principal Component Analysis

Explanation:

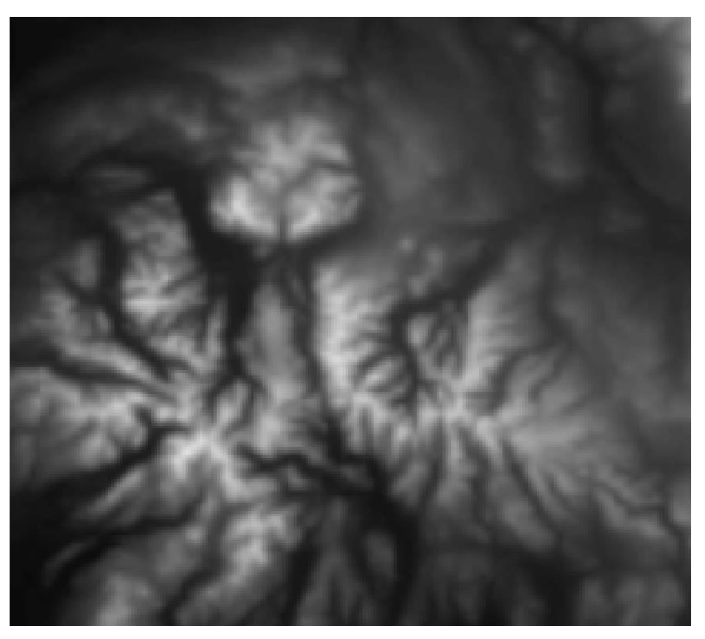
Principal Component Analysis (PCA) is a statistical method used for dimensionality reduction in data by transforming variables into a new set of uncorrelated components, (called principal values). The most significant information is captured, while minimizing loss of data variance.

Here you can see the impact the number of principal values has on the amount of variance explained. As the number of principal values are increased, they start to explain sequentially less of the variance. Therefore, we must determine how many of these values we want.

We plan to apply PCA to analyse satellite-captured LiDAR scans, converting elevation data into a Python array, and using the PCA algorithm to select the optimal number of principal components for the analysis, as demonstrated locally in Keswick.



[7]Principal Components Graph.



Procedure:

1. Standardise the range.

$$z = (x - \mu)/\sigma$$

2. Compute the covariance matrix, then remove data points with high similarity to preserve information and remove data.

$$Cov(X,Y) = \sum_{i=1}^{n} \frac{(X_i - \overline{X})(Y_i - \overline{Y})}{n-1}$$

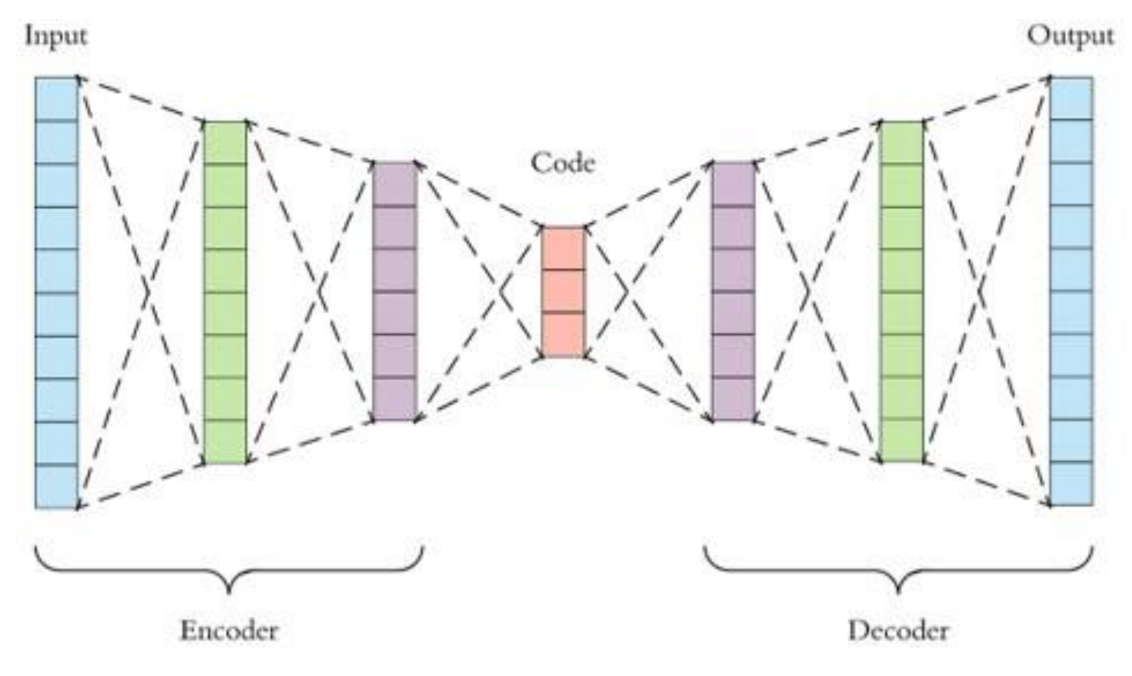
$$\begin{bmatrix} Cov(x,x) & Cov(x,y) & Cov(x,z) \\ Cov(y,x) & Cov(y,y) & Cov(y,z) \\ Cov(z,x) & Cov(z,y) & Cov(z,z) \end{bmatrix}$$

NY13NE
NY23NW
NY23NE
NY33NW
NY33NE
NY13SE
NY23SW
NY23SE
NY33SW
NY33SE
NY12NE
NY22NW
NY22NE
NY32NW
NY32NE
NY12SE
NY22SW
NY22SE
NY32SW
NY32SE
NY11NE
NY21NW
NY21NE
NY31NW
NY31NE
Keswick
Derwent Water
[8]LiDAR data - Keswick

Autoencoders

Process:

- An autoencoder is designed to reduce the dimensionality of data while minimising the loss of the key information.
- It does this by passing the data through multiple encoding layers which reduces the dimensionality of the data.
- The data is compressed as it passes through the bottleneck allowing only the most important information to pass through.
- The information is then reconstructed as close as possible back to its original state by using multiple decoding layers.
- At the bottleneck layer, the trained network will have compressed our terrain data into a low-dimensional latent space.
- By then randomly sampling from this space, we can get the decoder to produce completely new terrain.
- This new terrain will be similar to terrain the network was trained on and in addition very similar to particular terrains mapped to that region of the latent space.



[9]Autoencoder Theory - a visual representation of the neural network structure.

$$MSE = \frac{1}{n}\sum_{i=1}^{n} \iota(x_i, x_i')$$

[10]Mean Square Error loss function used in backpropagation.

Conclusion and Next Steps

Conclusion:

In summary, we have looked at the different methods used for terrain generation and carried out these methods to produce our own examples of how each method can generate terrain. We have explained how Principal Component Analysis can reduce the dimensionality of data while capturing the most important information and how this process is used in an important part of an autoencoder.

Next Steps:

- We are planning on writing code to convert the LiDAR scans into an array within Python, and then use a pre-existing library for Principal Component Analysis within Sci-Kit to reduce this data.
- We aim to train an auto-encoder on both the generated and real data by importing a machine learning library PyTorch, then creating new terrains from it.

References

[1] Minecraft Landscape - https://www.reddit.com/r/Minecraft/comments/r25c15/mountain_landscape_i_made_over_the_last_few_days
[2] Savitzky-Golay Filter - https://www.mathworks.com/help/signal/ref/sgolayfilt.html
[3] Diamond Square Theory - https://en.wikipedia.org/wiki/Diamond-square_algorithm
[4] Ken Perlin's Paper on Perlin Noise: An Image Synthesiser - ACM SIGGRAOH Computer Graphics, 19(3):287-296, 1985
[5] Perlin Noise Theory - https://en.wikipedia.org/wiki/Perlin_noise
[6] Perlin Noise Python Package - https://pypi.org/project/perlin-noise
[7] Principal Components Graph - https://link.springer.com/chapter/10.1007/978-3-031-04420-5_4
[8] LiDAR Data - https://www.data.gov.uk/dataset/f0db0249-f17b-4036-9e65-309148c97ce4/national-lidar-programme
[9] Autoencoder Theory - https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798
[10] Mean Square Error loss function - https://www.researchgate.net/publication/339480272_An_Overview_of_Deep_Learning_Architecture_of_Deep_Neural_Networks_and_Autoencoders