**1D Midpoint Elevation Generation**

My aim is to create and analyse an algorithm for the 1D midpoint method of terrain generation, demonstrating how the process varies for different base displacement values and a varying number of iterations of the algorithm.

**Concept behind the process**

1. Find midpoint coordinates
2. Displace these midpoints in the y direction
3. Add these y-displaced coordinates to the array of elevation of elevation coordinates
4. Repeat steps (1-3) t times, running the output from step 3 into step 1
5. Return the array of elevation coordinates (noisy)
6. Dampen the noise of these coordinates

It is these ‘de-noised’ coordinates that create the final elevation data.

**Aspects included**

* Start simply from two bounding x values, each at y=0.
* The elevation never goes below the x axis - i.e. if the displaced y is less than 0, then reverse the negative sign.
* The initial midpoint is always at the input displacement height. Input a randint for variation.
* Reproducibility - include a seed for the random number generator.
* An input number of iterations.
* At each repetition of finding midpoints, gradually scale down the y-displacement so that it produces a more smooth, natural result.
* Remove the large fluctuations of the coordinates (noise) by using the ‘Savitzky-Golay filter’ method. It basically finds line equations for the least square fit for incremental ranges of x and then records coordinates on the lines in an array.
* A plot that shows the noisy and quiet lines for each varying displacement ‘h’ value (where the noisy line plots are more transparent). The plots of different displacement values are colour coordinated for easy visualisation and analysis.
* Setup the coding as a module to simply call a function and return some elevation, where input parameters are: first x value, second x value, displacement h, number of iterations, seed
* The general function has relevant help if called for - help(function).

**Questions**

* How does increasing the displacement affect the elevation, with other parameters fixed?
  + My hypothesis is that it will obviously affect the maximum, but it will also affect the fluctuation.
* For a fixed base displacement h, is there a consistent number of iterations for a natural/smooth elevation as the bounds of x varies?
  + My hypothesis is that for each doubling of the magnitude between the bounds, the ideal number of iterations increases by 1.
* Does changing the number of significant figures of the random generated displacement affect the result?
  + My hypothesis is that it will be mostly the same with some slight change.

**Analysis**

For analysis, I have created 6 example plots: 2 seeds, 3 variations of iterations per seed, fixing the initial x values. The plots also include the noisy data (transparent) and the quiet data, colour-coordinated.

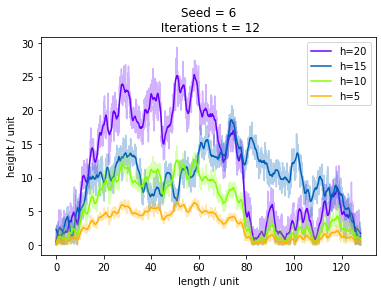
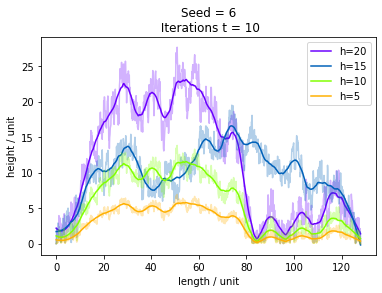
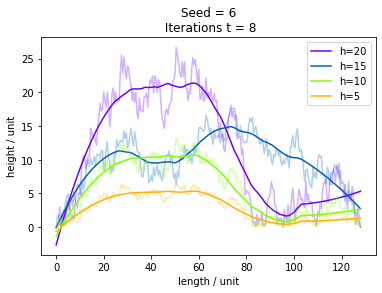
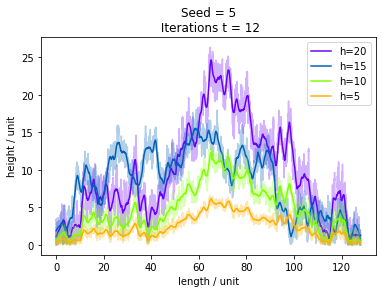
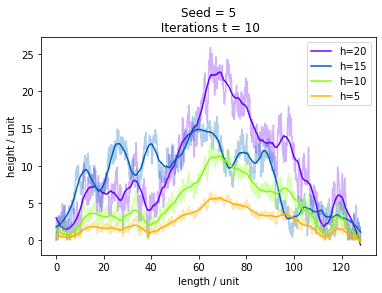
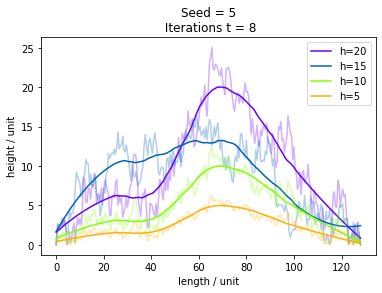
Looking at 3 plots of the same seed, it’s clearly depicted that the code is very sensitive to the number of iterations. Varying by -2 and +2 iterations from an arbitrarily discovered optimum of 10 produces elevation either very uninteresting or too overpowering (even when noise dampened).

Looking at an individual plot, the varied displacement functions similarly to my hypothesis: i.e. descending values produce much flatter, and lower plot. I found it interesting to note that some of these lines intersect (see the ‘seed=5 around x=50’ plot).

Changing the number of significant figures of the randomly generated displacement significantly alters the final product, contrary to my hypothesis.

Notes

The function also works if x0 > x1; however a different plot is produced from the same seed.

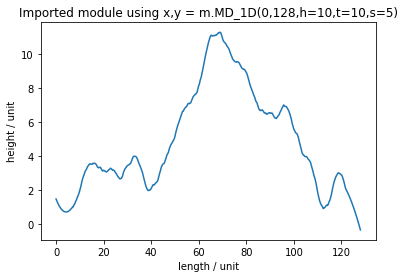
****

**Generalised Module**

I have created an importable module to include in other coding. It simply takes in the lower x bound, upper x bound, base height, number of iterations and seed. The function returns two arrays, one for the xs and one for the ys, to be used easily as a matrix. For example:

import Mid\_Dis\_1D\_Module as m

x,y = m.MD\_1D(0,128,10,10,5)

****

**Comparison Code**

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.cm as cm

import random

from scipy.signal import savgol\_filter

#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=20,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 so that to lessen 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):

D = M

#if its the first set of coordinates, set it to be the height

if len(M) == 1:

D[0][0][1] = h

else:

for i in range(0,len(M)):

D[i][0][1] = M[i][0][1] + random.randint(-n\*int(h),n\*int(h)) / (n\*z)

#terrain always greater than 0 i.e. above sealine

if D[i][0][1] < 0:

D[i][0][1] = -D[i][0][1]

#print("displaced midpoints",D)

return D

#Add the displaced coordinates to the original input array

def tot\_A(A,D):

B = []

j = 0

k = 0

while j < len(A) or k < len(D):

if j < len(A):

B.append(A[j])

j += 1

if k < len(D):

B.append(D[k][0])

k += 1

#print("new array",B)

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)

z=z+1

A=B

#Return an array containing a sequence of elevation coordinates

return(B)

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

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

seed = 5

t = 10

#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 - data, The length of the filter window i.e. # of coefficients,

#the order of the polynomial used to fit the samples

#Basically plots a least square fit to the data repeatedly for different windows of x

quiet\_y = savgol\_filter(E\_y\_coords, 101, 2)

#Plot the raw noisy coordinates, then the smooth quiet coordinates

plt.plot(E\_x\_coords,E\_y\_coords, color=cm.prism(h), alpha=0.3)

plt.plot(E\_x\_coords, quiet\_y, color=cm.prism(h), label="h={}".format(h))

#Plot features

plt.title("Seed = {0}\n Iterations t = {1}".format(seed,t))

plt.xlabel("length / unit")

plt.ylabel("height / unit")

plt.legend()

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A computer screen with text

Description automatically generatedA computer screen with text

Description automatically generated