Computer Science Extended Essay

Investigating factors of Noise generations in Perlin Noise for terrain generation through the videogame industry.

Research Question:

How does changing parameters (amplitude, frequency, and octaves) influence the visual appearance of terrains in Minecraft-like-worlds, as simulated using Perlin Noise in a Python program?

Rodrigo Blanco Maldonado

May, 2024

Word Count: 3994

Table of Contents

Table of Contents ii

1 - Introducing Procedural Noise - 1 -

2 - Theoretical Background Information - 2 -

3 - Methodology - 4 -

3.1 - Secondary Research Methodology: - 4 -

3.2 - Primary Research Methodology: - 5 -

4 - Secondary Research Conclusion on Parameters - 5 -

4.1 – Seed: - 6 -

4.2 - Octaves: - 6 -

4.3 - Amplitude: - 7 -

4.4 - Frequency: - 8 -

4.5 - Parameters Combined in Perlin Noise: - 9 -

5 - Researching Minecraft - 10 -

5.1 - Minecraft Data: - 13 -

6 - Primary Research, Own Experiment - 15 -

6.1 - Hypothesis: - 15 -

6.2 - Starting Parameter Data Sets for My Experiment: - 16 -

6.3 - Experiment setup: - 16 -

6.4 - Noise Function Variables: - 17 -

Independent Variables: - 17 -

Dependent Variables: - 17 -

Control Variables: - 17 -

6.5 - Minecraft Experiment Procedure: - 18 -

7 - Experiment - 20 -

7.1 - Frequency experiment: - 22 -

7.2 - Amplitude experiment: - 23 -

7.3 - Octaves experiments: - 24 -

First Trial – Octaves = **3**: - 24 -

Second trial – Octaves = **4**: - 25 -

Third trial – Octaves = **5**: - 26 -

Fourth trial – Octaves = **6**: - 27 -

7.4 - Octaves results: - 28 -

8 - Conclusion to my Research Question - 28 -

9 - Evaluation .... - 29 -

9.1 - Limitations .... - 30 -

9.2 - Further research .... - 30 -

Bibliography .... - 31 -

Appendix .... - 34 -

Graphs from primary research: .... - 34 -

Graphs for Octaves experiment: .... - 38 -

Program for primary research experiment: .... - 46 -

1 - Introducing Procedural Noise

In the scale of computer graphics, simulation, and content generation, there has been a pursuit for randomness, realism, and naturalness. Procedural Noise defines an algorithm able to generate noise patterns procedurally, or in other words, infinitely. The algorithms available for developers belong to one of three categories: Lattice Gradient Noise (LGN), Explicit Noise (EN), or Sparse Convolution Noise (SCN).[1]

Perlin Noise is a type of LGN, and more specifically it's a "gradient noise" which uses "random vectors".[2] Due to the production of vectors, it stands out for its ability to mimic the patterns and irregularities found in nature. Other alternatives such as "Lattice Noises" and "Simplex Noise"[3] lack the use of vectors, hence when generating textures, the result displays less smooth or coherent. Making Perlin Noise the best option for the gaming industry.

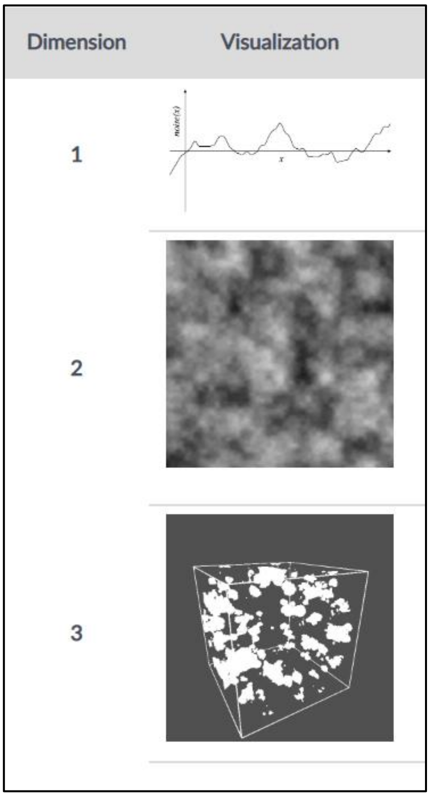
This investigation can have a significant influence on my understanding of game design. My motivation lies in revealing the depths of how this algorithm can bring life into virtual environments by providing creative freedom to generate unique and dynamic worlds, hence my initial research question is **"How does changing parameters for Perlin Noise influence the visual appearance of videogame terrains"**. I later refined this question to make it more focused.

The videogame industry is growing exponentially. There is a demand for more realistic graphics, that can be developed with the increasing technological power of computer systems to attract more players and producing memorable gaming experiences. Perlin Noise is important as it reduces the need for storage by generating terrains and textures only when necessary. This leads to a significant increase in the audience able to render videogames with infinite terrain generation.

[1] (*Procedural Noise Categories*)
[2] Ibid
[3] Ibid

2 - Theoretical Background Information

The algorithm was developed by Ken Perlin to generate procedural textures in 1983. This led him to win the “Technical Achievement Award from the Academy of Motion Picture Arts and Sciences”.[4] It revolutionized the way textures, landscapes, and natural phenomena were created. Perlin Noise is a type of gradient Noise that can be used to generate “smooth randomness”.[5] Unlike purely random Noise, Perlin Noise produces coherent patterns that resemble features seen in nature. The Noise can be generated in one or more dimensions; hence it has various appliances in different industries such as film production and videogame development.



Dimension	Visualization
1	
2	
3	

Figure 1 – representation of Noise in different dimensions[6]

Ken Perlin published his early research on November 3rd, 1985, where he stated the functionality of Perlin Noise. The algorithm is based on calculations, which when merged together they create the pseudo-randomness, giving the name of

[4] (Perlin *Noise and Turbulence*)
[5] (Hirnschall *Perlin noise: What is it, and how to use it.*)
[6] Ibid.

a “Pseudorandom” algorithm.[7] However, it gives the impression to be naturally random, as the human interpretation does not let the eye perceive the connection between the inputs and the outputs. Pseudorandom means that, by inputting the “same parameter twice, it produces the same number twice”, this is important as otherwise it would “produce nonsense”.[8]

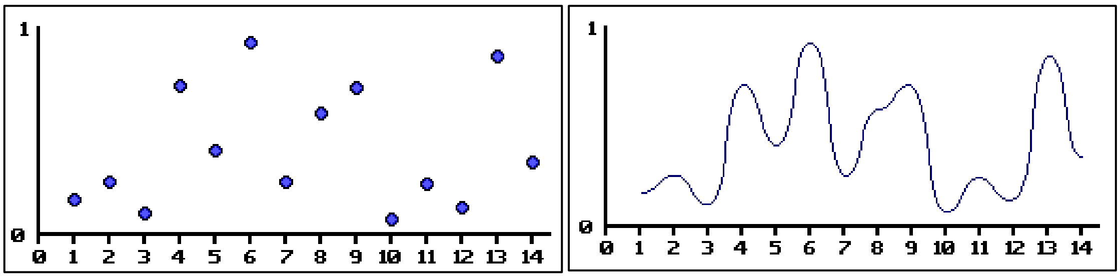


Figure 2 – Example of a procedural noise function[9]

The procedural noise function shown above creates a “random value between 0 and 1”. Then it “smoothly” interpolates between the integer x values to “define a continuous function”.[10]

The algorithm allows a control over the generated terrains and textures, permitting an adjustment in variables such as effects of rotation, scaling, and translation.[11] In videogames, altering Noise parameters allows developers to consistently generate random terrains and textures, offering new experiences to players. Additionally, it provides a tool that constructs a wide range of natural phenomena, including clouds and water droplets by using procedural animation, adding depth and realism in the virtual environment. In Minecraft specifically, it allows the generation of different biomes (regions in the world with distinct characteristics and features), each having a different set of parameters.

[7] (Biagioli *Adrian's soapbox*)
[8] (Elias *Perlin Noise*)
[9] Ibid
[10] Ibid
[11] (Perlin *An image synthesizer*)

A decade later, K. Perlin improved his Noise algorithm, upgrading the graphics and procedures. Perlin effectively eliminated visible edges and clumping effects, resulting in smoother and more realistic graphical textures. These improvements not only increase the visual upgrading of generated terrains and textures, but also add to a computationally faster algorithm. This means that game developers can now achieve more detailed and immersive environments without sacrificing performance.[12]

3 - Methodology

3.1 - Secondary Research Methodology:

The first aim is to understand and comprehend the functioning of Perlin Noise. I will read the documents “An image synthesizer” and “Improving noise” released by Ken Perlin. The reliability of these documents is certain, as they were published by the developer of Perlin Noise.

The second aim of this research is to conclude how parameters affect the quality of the content generated by Perlin Noise, as well as finding out if there is any major parameter that affects the noise more than others. To accomplish this, first I will examine the globally known videogame Minecraft (v1.16.5), which relies entirely on Perlin Noise for terrain generation. Then, I’m planning on researching how each individual major parameter affects the noise algorithm.

By inspecting the characteristics of biomes in Minecraft, I am hoping to extract my initial data set for my own experiment.

These resources are reliable as they have been developed by professionals and Minecraft is widely played by millions of users.

[12] (Perlin *Improving noise*)

3.2 - Primary Research Methodology:

After my secondary research on Minecraft, I will learn how to write a program in Python v3.11 to replicate a Minecraft world by using Perlin Noise and a 3D engine. For this I will watch YouTube tutorials to gain knowledge on how to generate the terrains.

The tutorials will be tested to fit the purpose of changing the inputs given to the noise. Hence, the code will be correct and appropriate for the research.

After finding the major parameters during my secondary research, I will experiment with them in my code to analyse the effect the change has on the quality of the generated terrain.

To measure the quality of the Minecraft terrain and my own generated terrains, I will establish 5 evenly spaced lines through the terrains to have a controlled experiment. These lines will be used to collect and analyse data for each terrain. The method is explained later in more details.

4 - Secondary Research Conclusion on Parameters

I observed that the seed plays a vital role in generating the unique random landscapes of Minecraft. It's a 64 bit number that is randomly generated.[13]

I also concluded that there are other significant parameters that can be controlled and manipulated to achieve desired outputs. It seems that the seed, octaves, amplitude, and frequency are specifically significant, while other parameters have a relatively smaller impact, and can be considered negligible. For example, the parameter persistence has a "value of 0.5, which is the default"[14] and "this is quite common. So common in fact, that many people don't even consider using anything else",[15] therefore I won't need to change it. If I am right, then by altering

[13] (Compton *How Minecraft generates massive virtual worlds from scratch*)
[14] (Bevins *Tutorial 4: Modifying the parameters of the noise module*)
[15] (Elias *Perlin Noise*)

octaves, amplitude, and frequency, I should be able to regenerate Minecraft terrains in my experiments.

Here's a breakdown of each major parameter and how the change in value changes the image quality:

4.1 - Seed:

The seed is the integer number that states the entire pattern of the generated Noise. In the same way that the DNA contains the pattern of human genes, the seed contains the pattern of the terrain generation. A seed has no correlation to the previous or next integer seed, hence creating an unpredictable randomness. The same seeds with equal additional parameters result in identical Noise patterns, no matter how many times the terrain is generated. Usually, a random seed value is used to achieve pseudo-randomness. In some cases, it’s necessary to create identical Noise patterns. For example, in competitions where participants compete in the same world, the same seed is used to generate the worlds. The seed, in Minecraft, can be a number from 0 to $2^{64} - 1$, as it contains 64 bits.

4.2 - Octaves:

States the number of simple Noise generators that are combined as layers to create the final result, making the terrains smooth or complex. Each additional octave adds finer details to the Noise pattern while also increasing complexity, which is shown by the gradient of the terrain, found by dividing the change in height y with the change in distance x. The greater the change in height for the same distance the higher the gradient and complexity.

This value usually goes from 1 to 6. This is due to the insignificant scale of the layers after 6 octaves. (*Figure 3*)

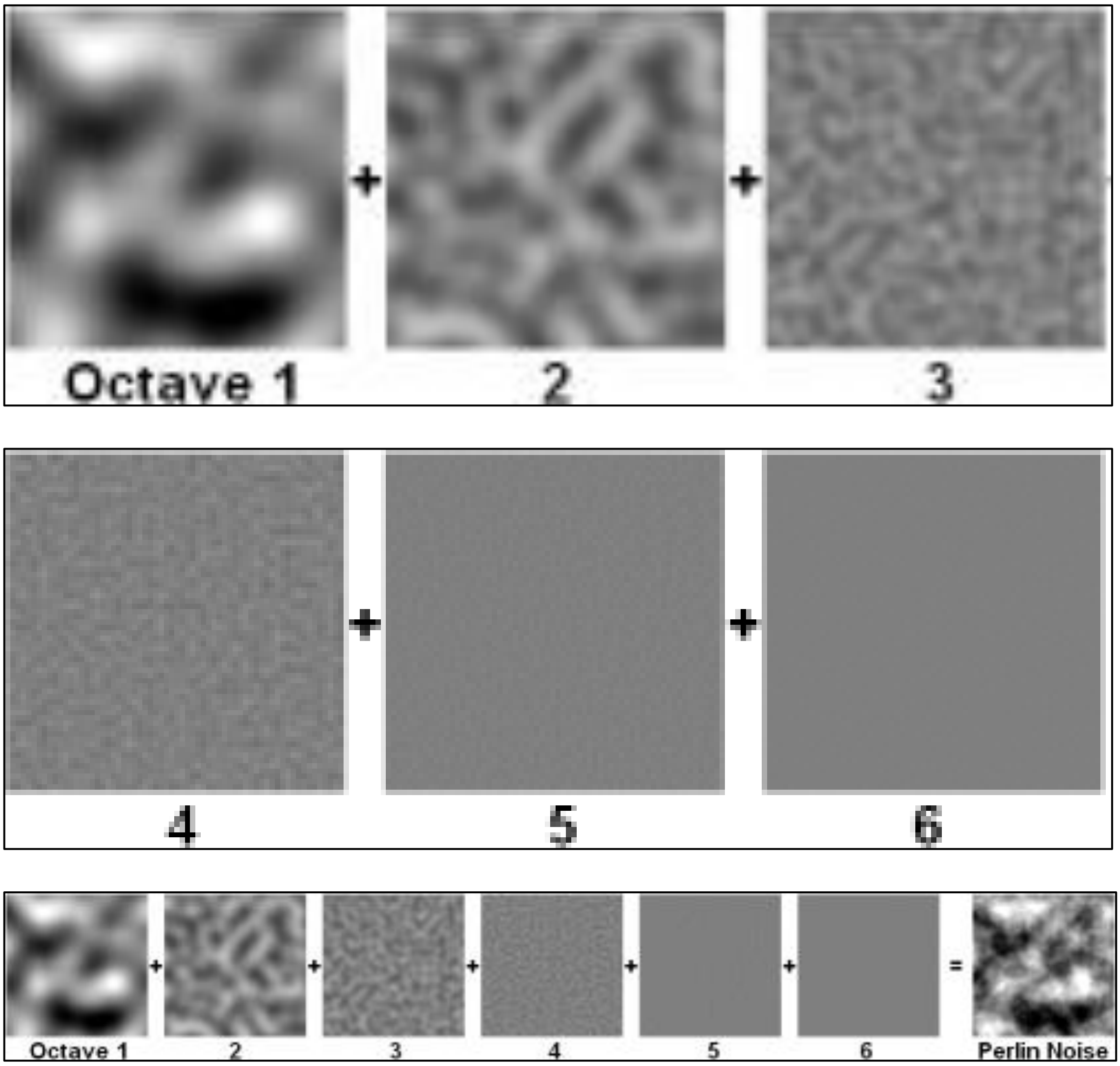


Figure 3 – How consecutive octaves have higher complexity and combine to form Perlin Noise.[16]

4.3 - Amplitude:

The amplitude can be called the "height" of the Noise wave and pattern, especially for the first octave. With subsequent octaves the amplitude decreases generating a varied Noise. This parameter consists of a vector acting vertically upon the y values, allowing the noise to grow and shrink along the axis. The value can go as high as the wanted height for the terrain. I checked the Minecraft version I will use, and the height y limit they use is 0-256, with sea level being at 62 blocks.

[16] (Bevins *Tutorial 4: Modifying the parameters of the noise module*)

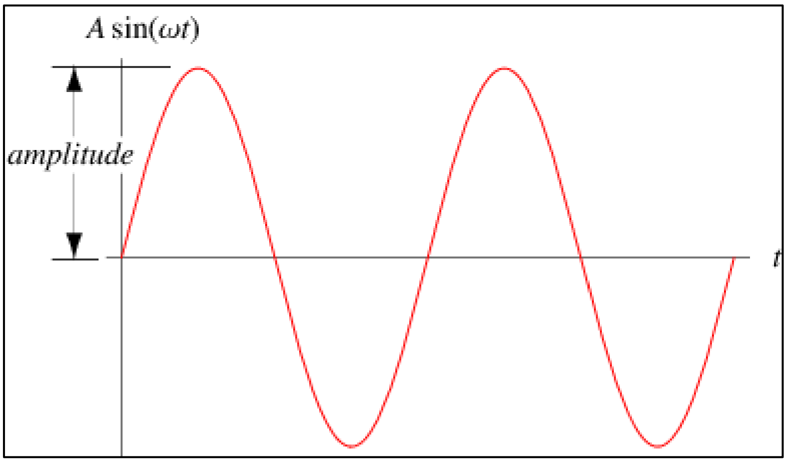


Figure 4 – A typical sin wave, describing the amplitude or height of the wave.[17]

4.4 - Frequency:

This parameter may be seen as the "wavelength" of the noise pattern and refers to how often the characteristics of the terrain repeats itself. With every following octave the frequency value will be higher to create finer and more precise details. The frequency is a vector with values for x and z that allows the noise to stretch or compress along horizontal axes. The frequency value is how often the terrain should repeat, and there's no boundaries for this value.

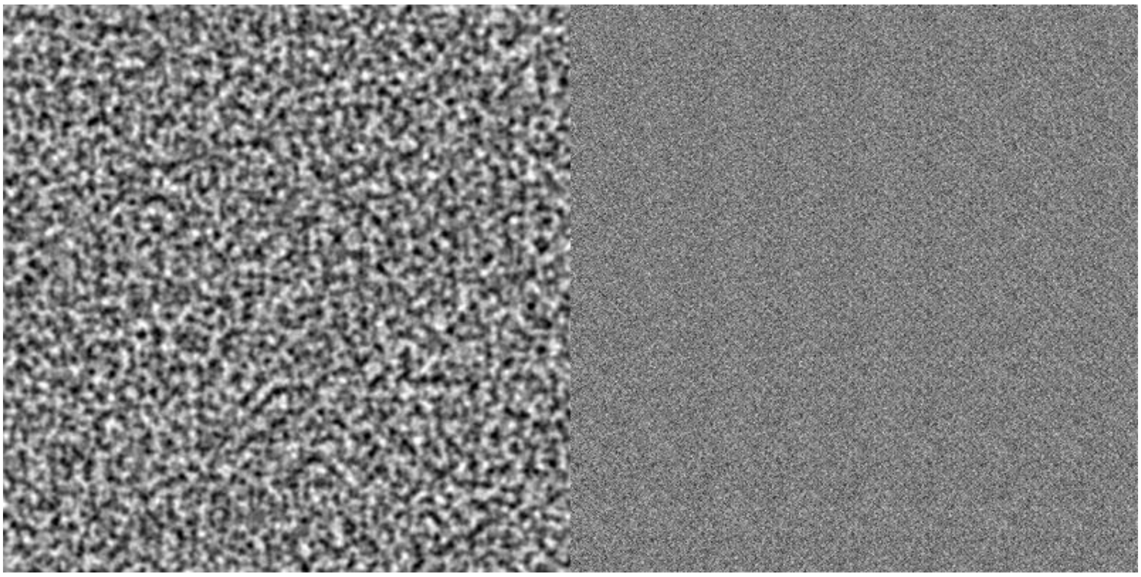


Figure 5 - Two different frequencies on the same noise. A higher frequency with repeating noise pattern is shown on the right.[18]

[17] (*Understanding some basic noise terms*)
[18] Ibid

4.5 - Parameters Combined in Perlin Noise:

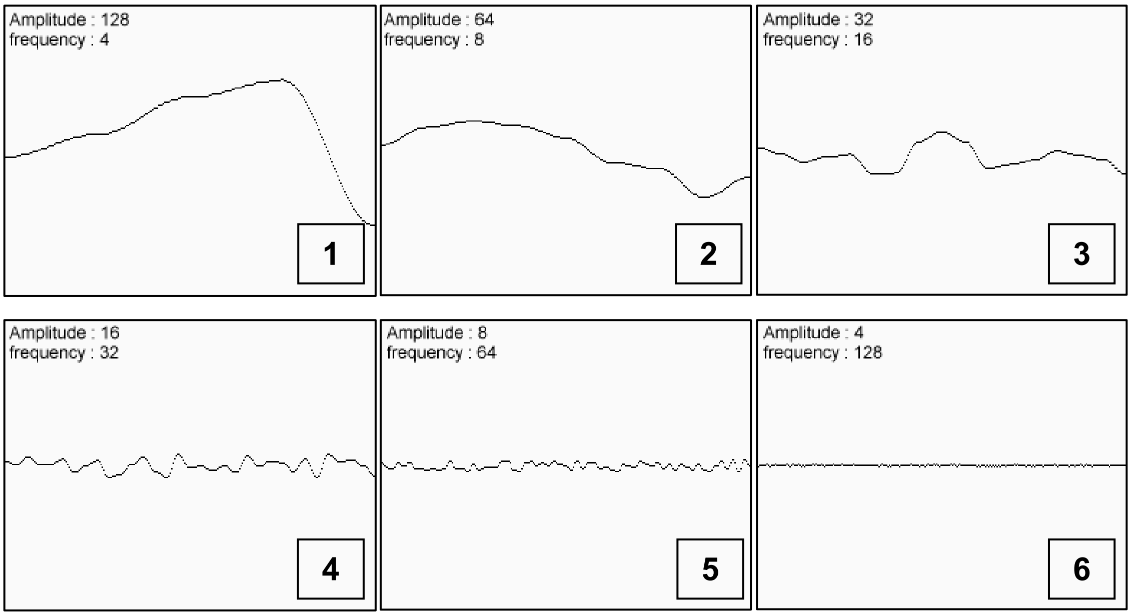


Figure 6 – Noise pattern in 1 dimension for labelled subsequent octaves from 1 to 6[19]

As seen in *Figure 6*, the initial parameters are amplitude:128 and frequency:4, which describe the noise for the first octave. For each additional octave the amplitude is halved, and the frequency doubled. After the six octaves, every noise function is added to give the final result shown in the figure below.

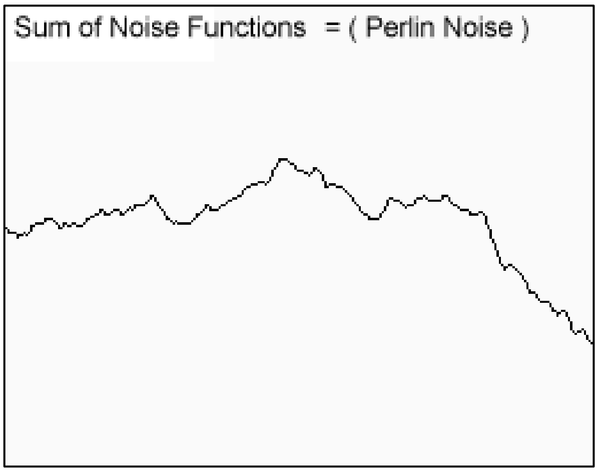


Figure 7 – Final output of Perlin Noise in 1 dimension[20]

[19] (Elias *Perlin Noise*)
[20] Ibid

The end result, as shown contains “large, medium and small variations”. Perlin Noise can display images and textures similar to a “mountain range”, as the ones seen in nature.[21]

5 - Researching Minecraft

Different biomes have different characteristics which are seen in every Minecraft world. For example, plains are typically flat, while mountains are always spiky. Therefore, in Minecraft’s code for Perlin Noise, there should be different combinations of parameters for every biome.

I chose the most recurrent and typical biome in Minecraft: plains. To analyse the biome, I created a new world. This world contains one unique biome that extends throughout the whole map. This allows me to analyse a greater area, without having to endlessly search for a suitable terrain in a common Minecraft world. I then proceeded to delimitate a perimeter around the section I will analyse.

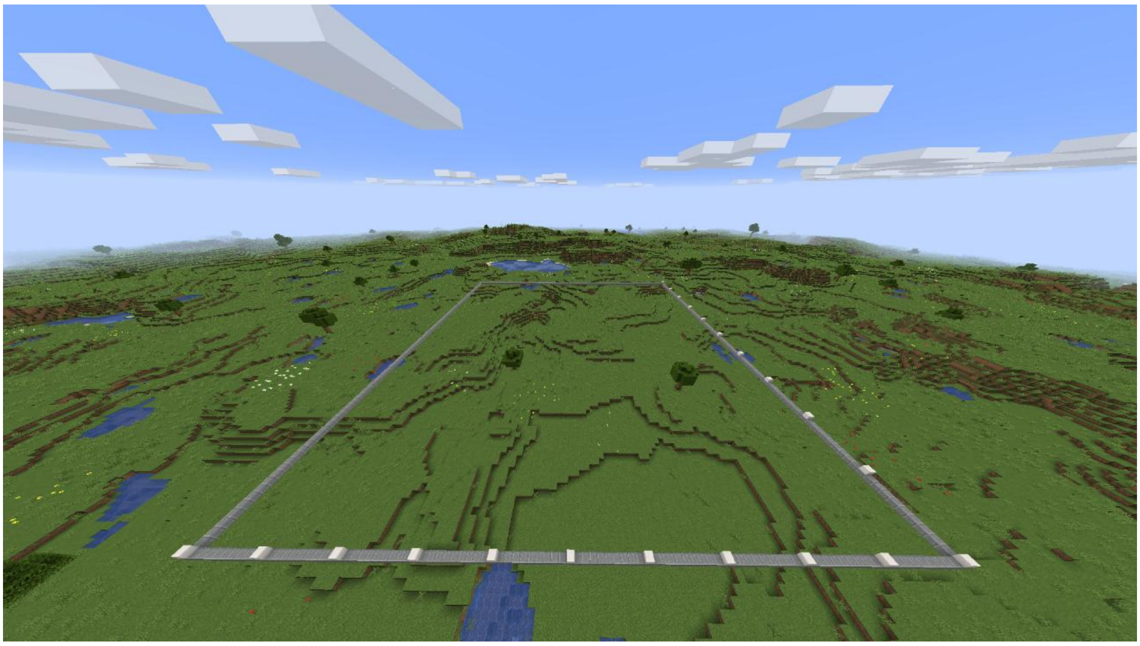


Figure 8 – Delimitated area to analyse Plains

[21] (Elias *Perlin Noise*)

The area shown above is a 100 x 100 blocks square, with white marks repeating every 10 blocks. When increasing the width of the generated terrain, the precision and reliability of the findings increases, due to the larger surface area been analysed. However, I then had to generate a terrain in my code with the same areas and my code is not well optimised as Minecraft and generating terrain takes a lot of memory space. So, I maximised the width with the computational load available in my personal computer system hardware.

I cleaned the terrain by eliminating any flowers and trees. I built 5 lines through the whole area. These lines will be used to gather data at 5 different points, followed by calculating the average for the sample data.

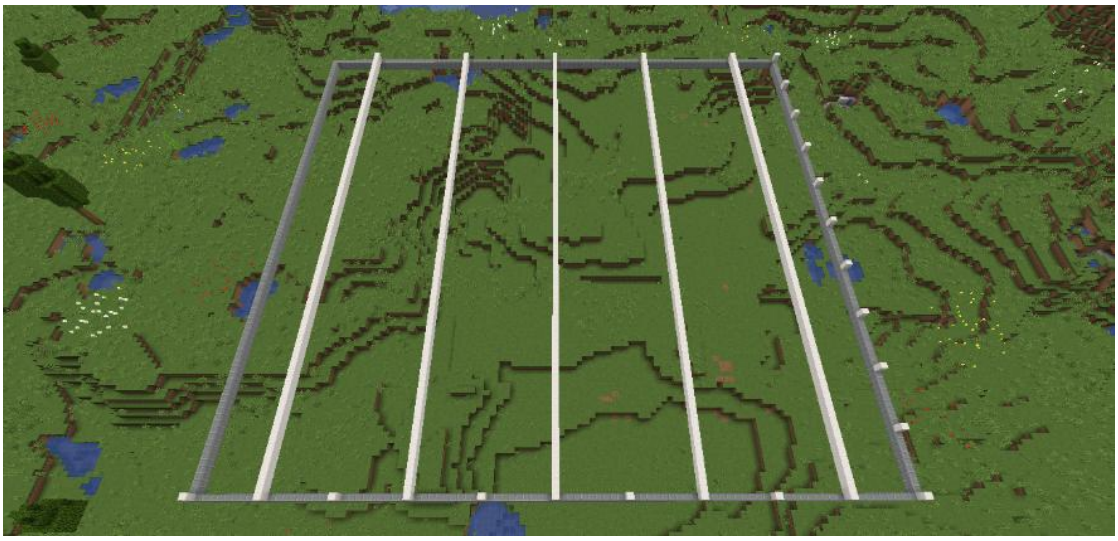
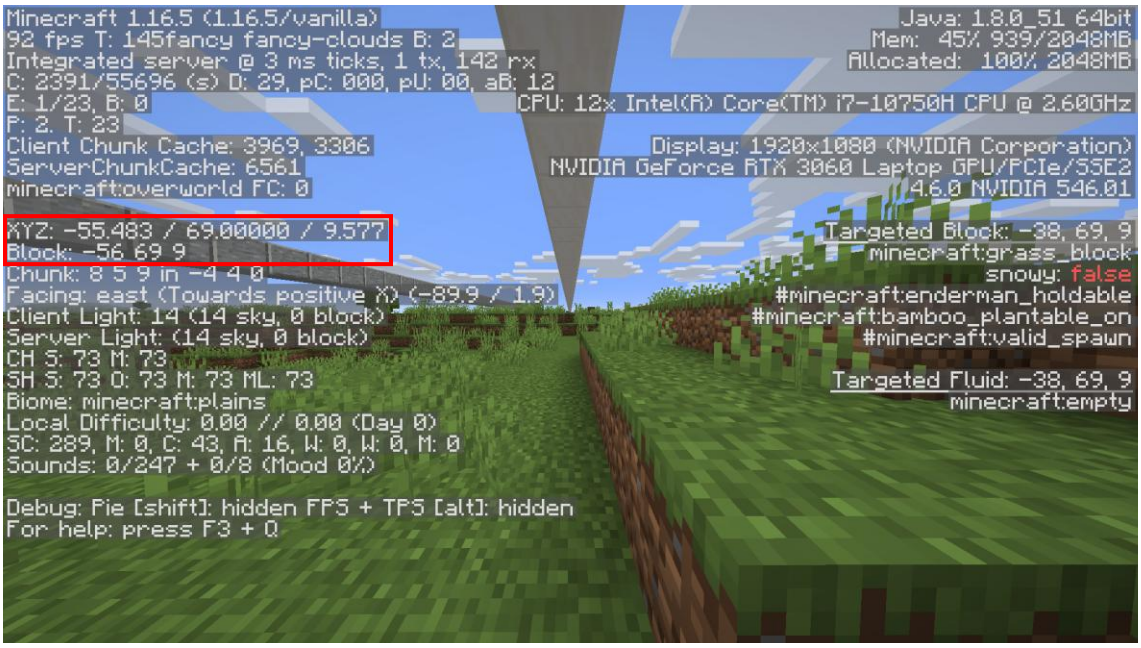


Figure 9 – Delimitated area to analyse Plains, cleaned with data lines

The data lines shown above will be referred as P1 to P5 from left to right respectively as seen in *Figure 9*. As a sample, only the first data line will be shown, this applies the whole research (see Appendix for lines P2 to P5 – Graphs 7-10). For every line, I recorded the y value, so height, against the horizontal position x of the block. My method to record the values was pressing the button F3, which shows the coordinates of the player and the block below, as shown in the red box in *Figure 10*.

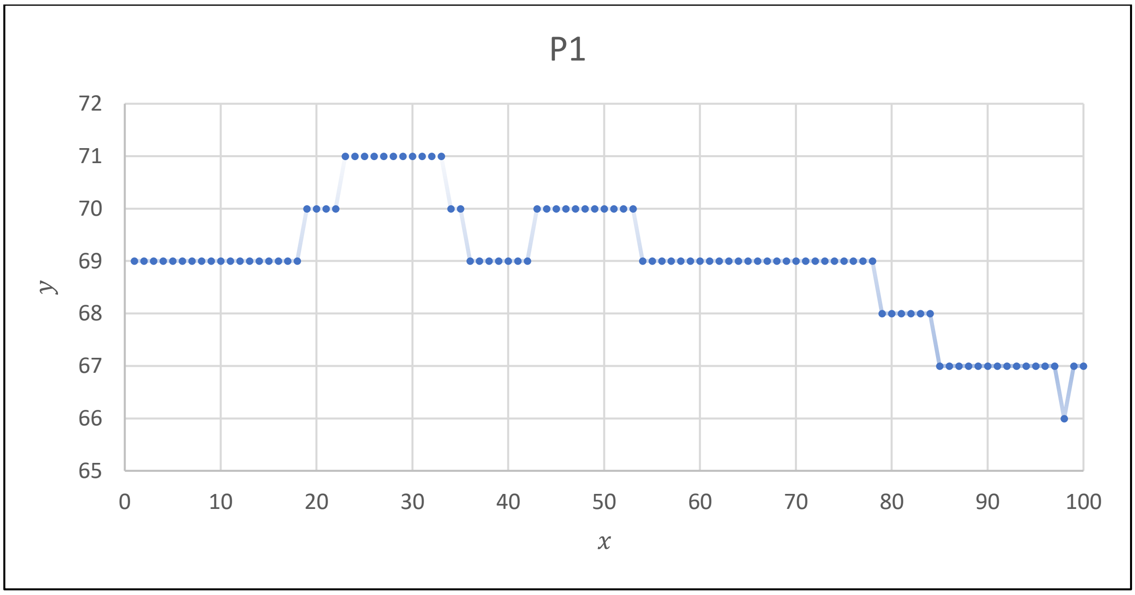


```
Minecraft 1.16.5 (1.16.5/vanilla)
92 fps T: 145fancy fancy-clouds B: 2
Integrated server @ 3 ms ticks, 1 tx, 142 rx
C: 2391/55696 (s) D: 29, pC: 000, pU: 00, aB: 12
E: 1/23, B: 0
P: 2. T: 23
Client Chunk Cache: 3969, 3306
ServerChunkCache: 6561
minecraft:overworld FC: 0

XYZ: -55.483 / 69.00000 / 9.577
Block: -56 69 9
Chunk: 8 5 9 in -4 4 0
Facing: east (Towards positive X) (-89.9 / 1.9)
Client Light: 14 (14 sky, 0 block)
Server Light: (14 sky, 0 block)
CH S: 73 M: 73
SH S: 73 O: 73 M: 73 ML: 73
Biome: minecraft:plains
Local Difficulty: 0.00 // 0.00 (Day 0)
SC: 289, M: 0, C: 43, A: 16, W: 0, W: 0, M: 0
Sounds: 0/247 + 0/8 (Mood 0%)

Debug: Pie [shift]: hidden FPS + TPS [alt]: hidden
For help: press F3 + Q
```

```
                          Java: 1.8.0_51 64bit
                       Mem:  45% 939/2048MB
                     Allocated:  100% 2048MB
CPU: 12x Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz

         Display: 1920x1080 (NVIDIA Corporation)
  NVIDIA GeForce RTX 3060 Laptop GPU/PCIe/SSE2
                          4.6.0 NVIDIA 546.01

                Targeted Block: -38, 69, 9
                    minecraft:grass_block
                            snowy: false
             #minecraft:enderman_holdable
          #minecraft:bamboo_plantable_on
                 #minecraft:valid_spawn

                Targeted Fluid: -38, 69, 9
                        minecraft:empty
```

Figure 10 – Sample image of how to gather data

From the coordinates I know the first block of line P1 is at $y = 69$. The height was recorded against the x value which went from 1 to 100, ignoring the x and z coordinates from Minecraft.



Graph 1 – P1 data for block position

The graphs plotted for the five lines simply show a side view of the surface layer of the terrain. This allows me to see how the terrain changes due to the noise function. Which means some of the characteristics and parameters of the noise can be inferred from these graphs. The characteristics to focus on are the peaks and troughs of the terrain, as well as the maximum and minimum height of the blocks. By analysing these characteristics, I am able to calculate an estimated value for the frequency and amplitude of the noise function.

5.1 - Minecraft Data:

I recorded the data from every data line graph (see Appendix) into the following table. The data includes the distance x of a first and second peak or a first and second trough. One of the two options was recorded depending on what was available in each graph. Additionally, I recorded the maximum and minimum height y of the data lines.

Data Line	First peak or trough	Second peak or trough	Maximum heigh	Minimum height
P1	39	98	71	66
P2	6	78	72	64
P3	50	100	72	68
P4	37	77	71	68
P5	35	91	71	67

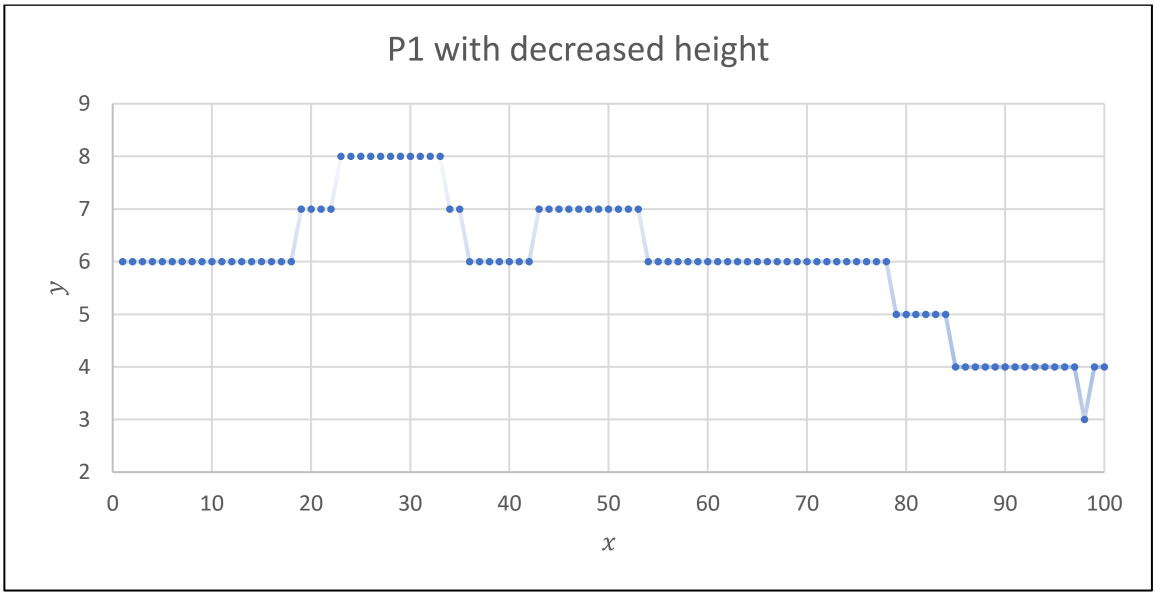
Table 1 – Raw data from data lines

With the data above I am able to find the difference between the peaks or throughs, and the heights. Therefore, I can calculate the exact values of the frequency and amplitude for every line. The average will be calculated by adding the values for the lines and dividing by 5, and will be rounded to the nearest integer. This is due to the blocks been on specific integer coordinates, making it impossible for blocks to be in decimal positions. The average will allow me to compare the surface areas of the Minecraft terrain to the ones I am going to create with my code.

Data Line	Frequency	Amplitude
P1	59	5
P2	72	8
P3	50	4
P4	40	3
P5	56	4
Average	55	5

Table 2 – Processed data from data lines

For a block, its y value states the number of blocks below, until it reaches the limit of height 0. So, I added the y value for every block to find the area of each data line. The code I am going to use for my own research will only generate the surface layer, making the minimum height 1, which doesn't match with Minecraft terrains. So, to solve the problem, I decreased the height of the blocks by 1 block less than the minimum height of the Minecraft terrain. This makes the new minimum height 1, which is consistent with the terrain generated by my code. From *Table 1* the lowest minimum height is $y = 64$ so all the blocks will be decreased by 63. (P2 to P5 – Graphs 11-14 in Appendix)



Graph 2 - P1 data for block position with decreased height

When repeating the process for every graph, I found the average value for area under the terrain surface.

Data Line	Area
P1	600
P2	487
P3	641
P4	682
P5	602
Average	602

Table 3 – Area for every data line with average

6 - Primary Research, Own Experiment

My aim will be to investigate the role of frequency, amplitude, and octaves on the Noise function. I am going to achieve this by replicating the Minecraft terrain by using my own code and changing these parameters. To start, I will use the main parameters identified in *Table 2*. By establishing and analysing lines as before, I will be able to compare terrains to each other and draw conclusions for my research question.

6.1 - Hypothesis:

My theory for Perlin Noise is that:

1 - Modifying the frequency will affect how often the terrain repeats.

2 - Changes in amplitude affect the height y of terrains.

3 - Changing the octaves varies the smoothness and complexity of the terrain.

4 - The 3 above are the main parameters that affect the noise generation significantly.

6.2 - Starting Parameter Data Sets for My Experiment:

After investigating the Minecraft terrain, the optimum values for which my code should reproduce Minecraft plains biome generation are listed below in *Table 4*. From my Minecraft investigation, I was not able to come up with a number for the octaves. Therefore, I will start with the lowest value of 1, and go up to 6 throughout the experiment. The best octaves value will be the one that gives me the closest area to 602 as stated in *Table 3*.

Parameter	Noise Function value
Frequency	55
Amplitude	5
Octaves	1

Table 4 – Summary of 5.1 data

6.3 - Experiment setup:

I wrote a Python v3.11 code that includes 4 files: Main, Terrain, Perlin, and Perlin_Noise (see Appendix).

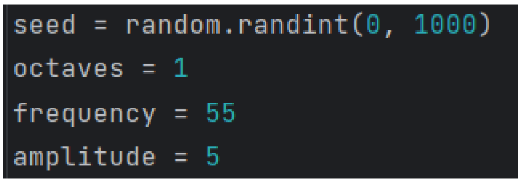
File	Purpose
Main	– Creates app and player properties. – Continuously checks the height of terrain to update the y value of the player when going up and down the terrain.
Terrain	– Generates every block in the terrain. – Adds specific textures to blocks depending on the height. – Saves heights of the 5 data lines into a spreadsheet.
Perlin	– Contains the parameters for the noise function.
Perlin_Noise	– Contains the algorithm for Perlin Noise.[22]

Table 5 – Summary of each file's purpose

[22] (Red Hen Dev *perlin_module.py*)

6.4 - Noise Function Variables:

Independent Variables:



```
seed = random.randint(0, 1000)
octaves = 1
frequency = 55
amplitude = 5
```

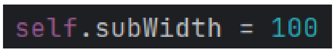
Figure 11 – Parameters in the algorithm

These are the parameters that will be changed in the code during the experiment. The seed will be produced randomly for the different types of experiments but will be kept the same within each type of experiment. It will be created by using a random integer (0 to 1000) from the random python library. This is done to mitigate biased results.

Dependent Variables:

The variable I am going to measure from the data lines is the area under the terrain. By changing the independent variables, the area will change. This leads to an impact on the quality when compared to the Minecraft terrain.

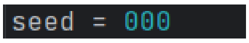
Control Variables:



```
self.subWidth = 100
```

Figure 12 – Variable that is not changed

This is the width of the terrain, which will be kept constant at 100 blocks throughout the experiment. Hence, the surface area analysed for Minecraft and my code is the same.



```
seed = 000
```

Figure 13 – Parameter seed to input a wanted value within experiments

The seed will be constant within each type of experiment to avoid randomicity.

6.5 - Minecraft Experiment Procedure:

Frequency and amplitude experiments:

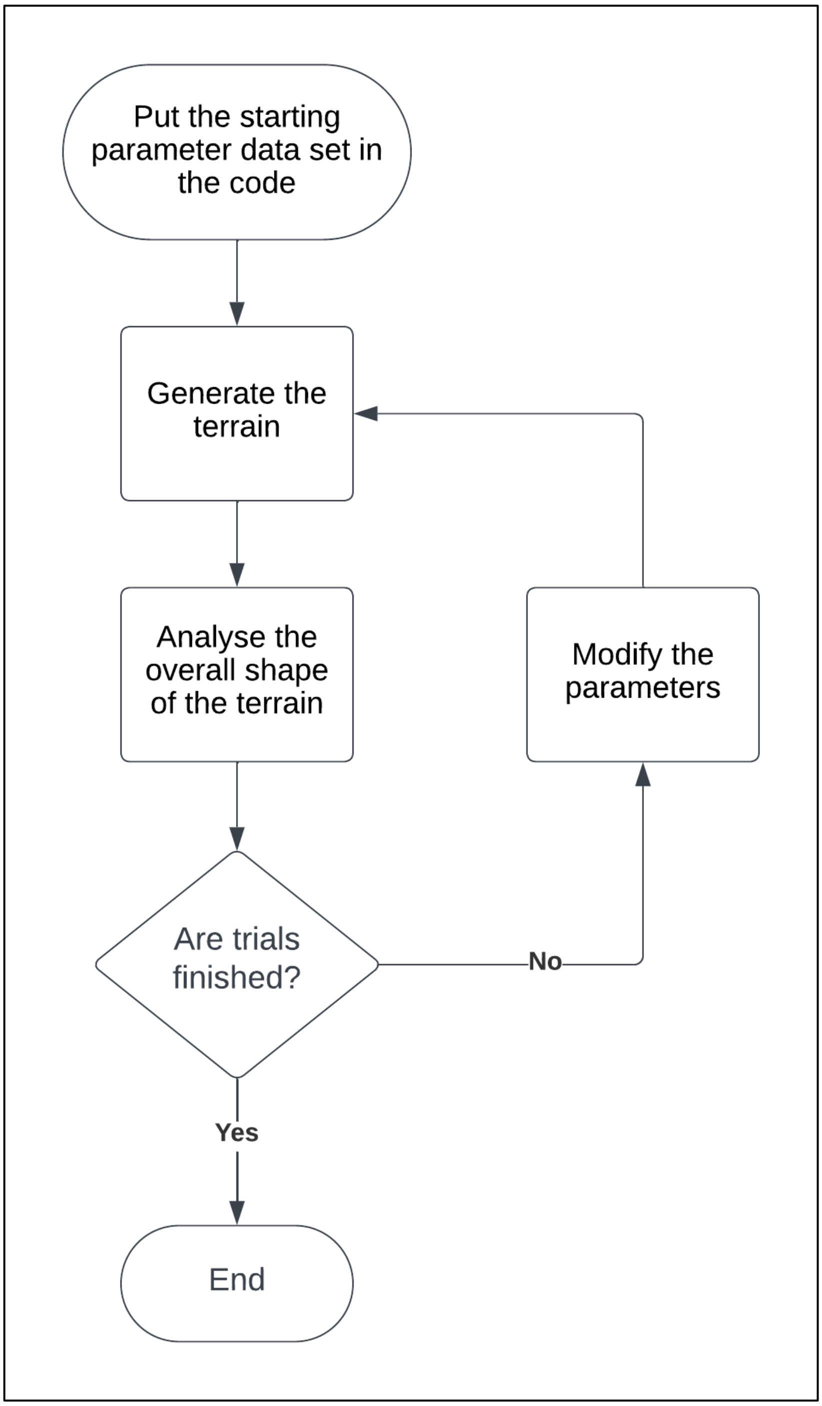

Put the starting parameter data set in the code
Generate the terrain
Analyse the overall shape of the terrain
Modify the parameters
Are trials finished?
No
Yes
End

Figure 14 – Procedure for analysing parameters

Octaves and Minecraft replication experiment:

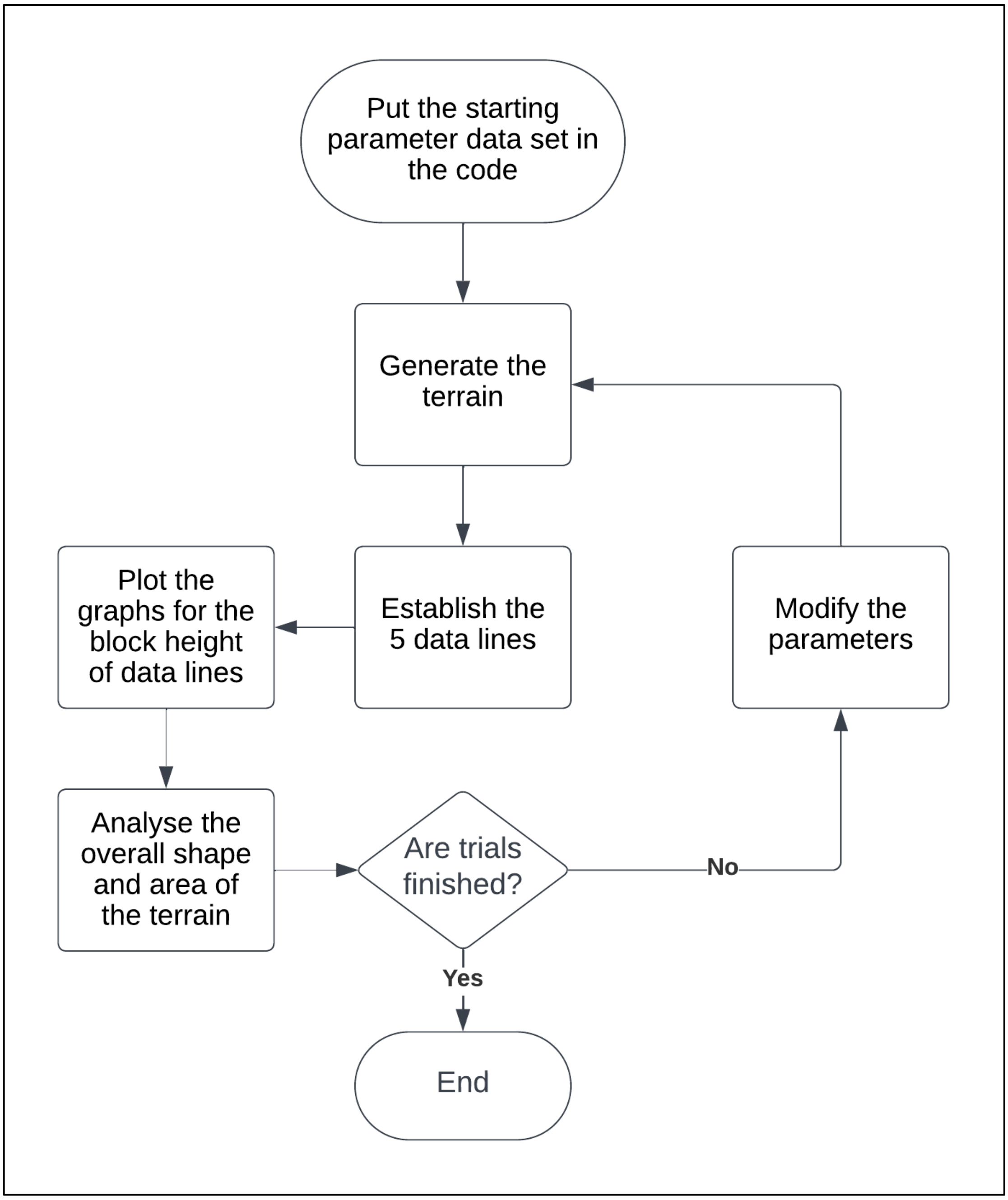

Put the starting parameter data set in the code
Generate the terrain
Establish the 5 data lines
Plot the graphs for the block height of data lines
Analyse the overall shape and area of the terrain
Are trials finished?
No
Modify the parameters
Yes
End

Figure 15 – Procedure to analyse octaves while replicating Minecraft

7 - Experiment

To set up the experiment, I will create a sample terrain with two views using the initial data set.

World view: to walk on the terrain, having similar block textures to Minecraft.

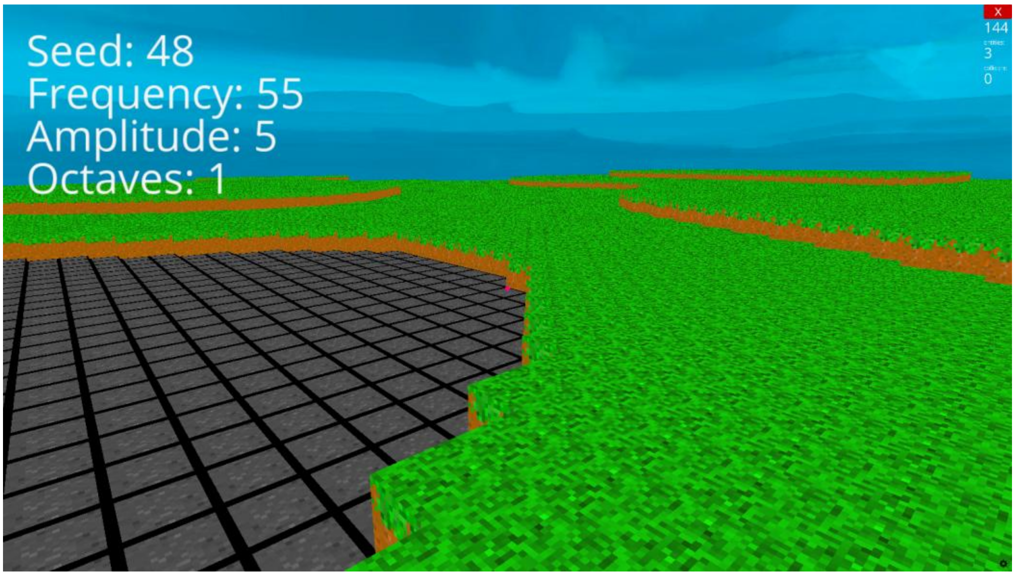

Seed: 48
Frequency: 55
Amplitude: 5
Octaves: 1

Figure 16 – World sample image

Altitude view: looked from above, with solid colour block textures representing the height layers.

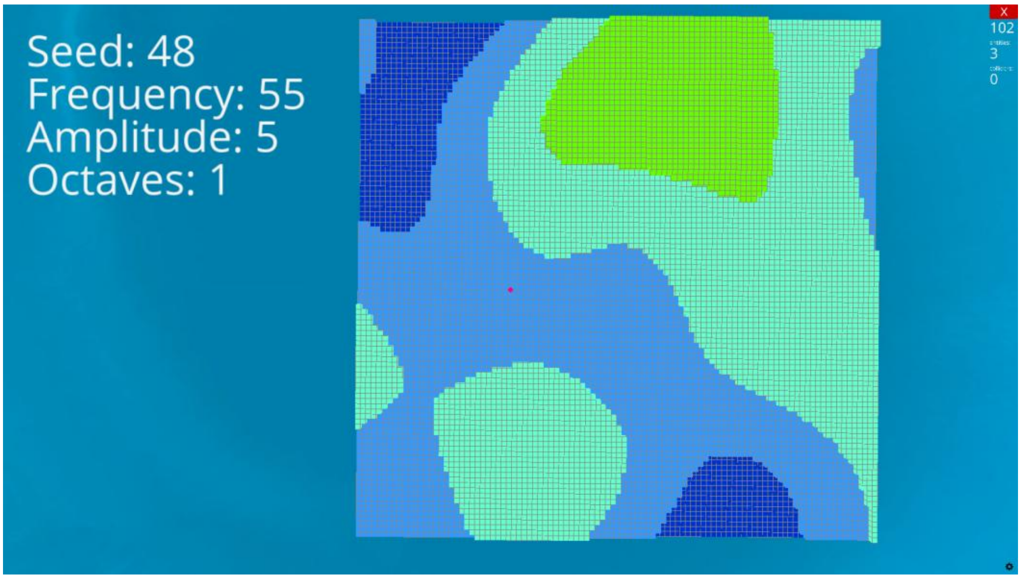

Seed: 48
Frequency: 55
Amplitude: 5
Octaves: 1

Figure 17 – Altitude sample image

Both terrains have the block textures related to the height of the block. For example, in the first world, the grass block appears at $3 \leq y < 6$.

Block type	Height
Snow	$9 \leq y$
Snowy dirt	$6 \leq y < 9$
Grass	$3 \leq y < 6$
Stone	$1 \leq y < 2$

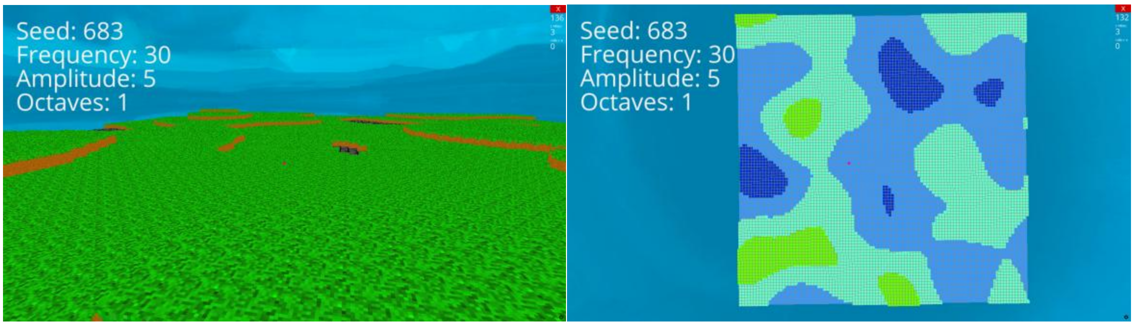
Table 6 – World block texture height values

Block type	Height
Red	$10 \leq y$
Orange	$8 \leq y < 10$
Yellow	$6 \leq y < 8$
Green	$y = 5$
Aqua	$y = 4$
Light blue	$y = 3$
Dark blue	$y = 2$
Purple	$y = 1$

Table 7 – Altitude block texture height values

7.1 - Frequency experiment:

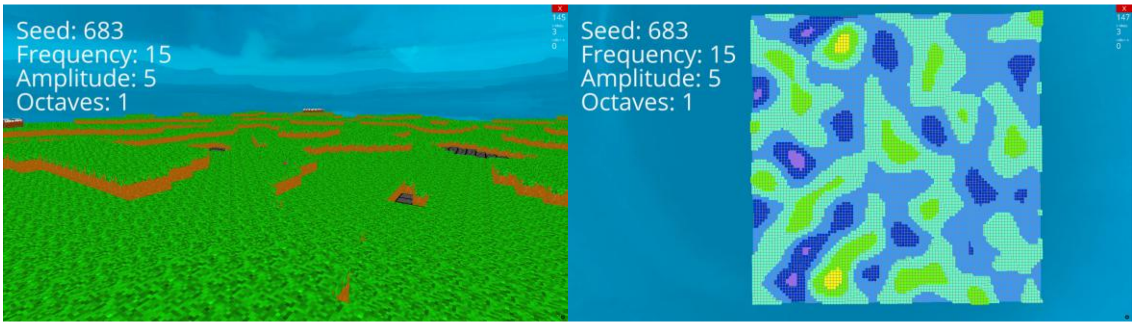
For the initial data set there was only one peak in the terrain generated, as there is just one green area of height 5 in *Figure 17*. To increase the number of peaks, I will make the frequency smaller twice, from 55 to 30 and then to 15. Then I will compare the results for both terrains.



Seed: 683
Frequency: 30
Amplitude: 5
Octaves: 1

Seed: 683
Frequency: 30
Amplitude: 5
Octaves: 1

Figure 18 – Terrain for frequency 30



Seed: 683
Frequency: 15
Amplitude: 5
Octaves: 1

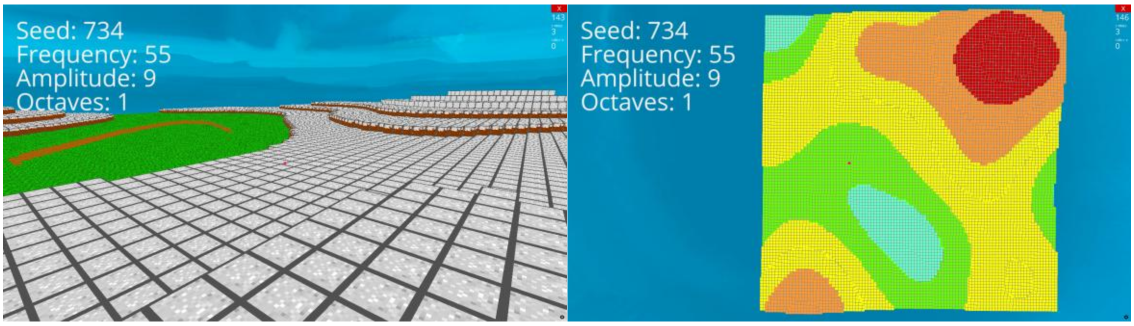
Seed: 683
Frequency: 15
Amplitude: 5
Octaves: 1

Figure 19 - Terrain for frequency 15

As seen above, when the frequency decreases the number of green areas increases. This means that peaks get closer together and the terrain repeats itself more often with lower frequencies. So, it seems that the frequency has a significant effect on the terrain.

7.2 - Amplitude experiment:

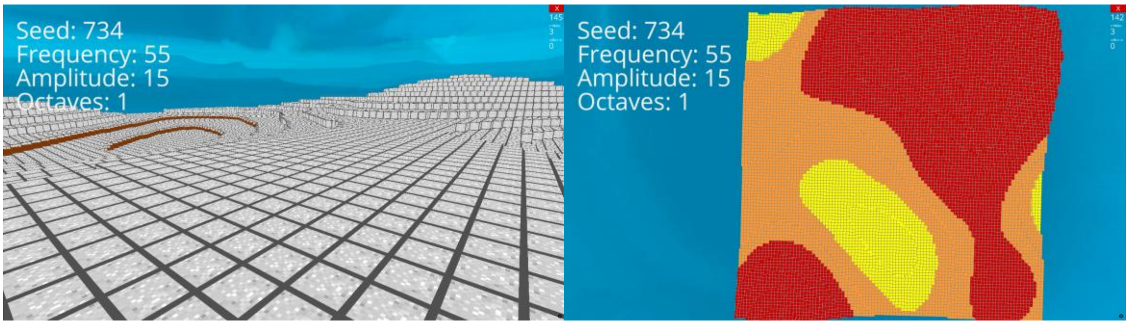
When generating the first terrain, the maximum height was 5, due to the green seen on the peak. However, as shown on *Figure 19* during the frequency experiment, there are some yellow blocks on the peaks. Hence, for amplitude 5 the maximum height is 6. To analyse amplitude, the value will be increased to 9, and then to 15.



Seed: 734
Frequency: 55
Amplitude: 9
Octaves: 1

Seed: 734
Frequency: 55
Amplitude: 9
Octaves: 1

Figure 20 – Terrain for amplitude 9



Seed: 734
Frequency: 55
Amplitude: 15
Octaves: 1

Seed: 734
Frequency: 55
Amplitude: 15
Octaves: 1

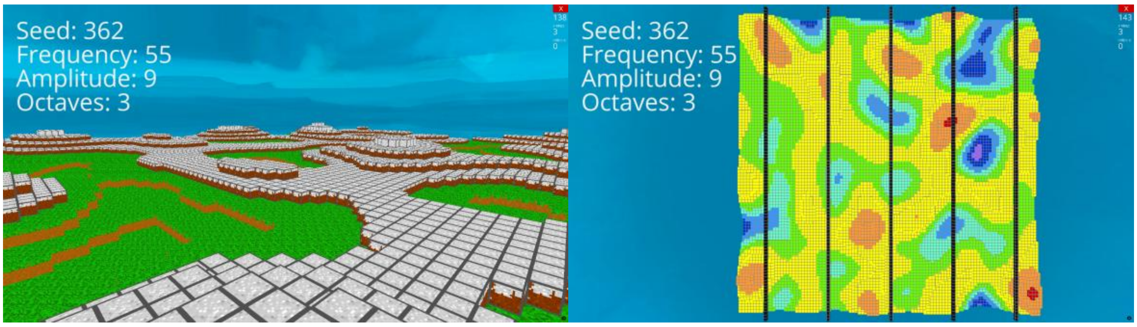
Figure 21 – Terrain for amplitude 15

From the amount of snow on top of the peaks and the highly increasing red colours, I can say that increasing the amplitude makes the maximum height greater. In *Figure 20* the most recurring colour is yellow, meaning that the average height of blocks is $6 \leq y < 8$. Similarly, from looking at the graphs for Minecraft data with decreased height, the most repeated heights are 6 and 7. Hence, a better value for amplitude is 9, so I will use this value for experiments.

7.3 - Octaves experiments:

When generating the terrain with the starting data set, (*Figures 16 and 17*) I realised it was smoother than the Minecraft world. As more complexity is needed, I will investigate octaves 3 to 6. The data lines will be shown in black blocks and will be referred as C1 to C5. I am going to **keep the seed the same** to avoid bias.

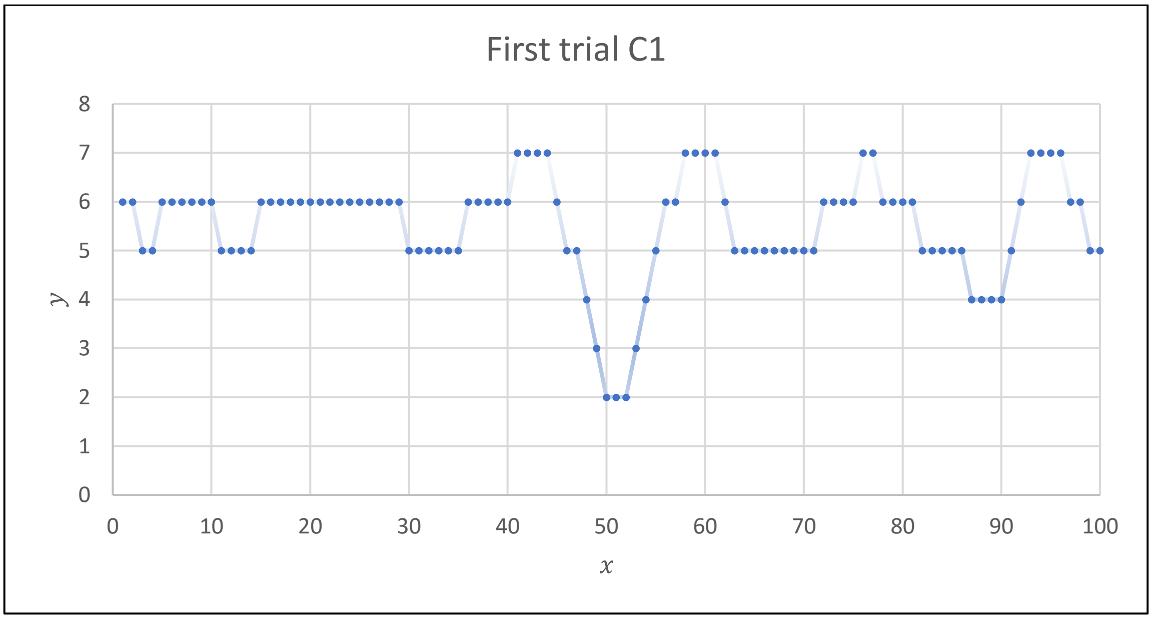
First Trial – Octaves = 3:



Seed: 362
Frequency: 55
Amplitude: 9
Octaves: 3

Seed: 362
Frequency: 55
Amplitude: 9
Octaves: 3

Figure 22 – Terrain for Octaves experiment first trial

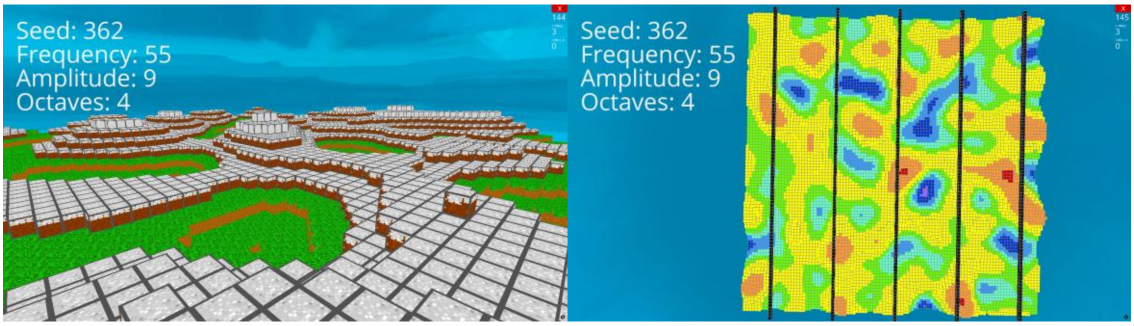


Graph 3 – C1 for Octaves experiment first trial

Data Line	Area
C1	336
C2	348
C3	363
C4	398
C5	340
Average	357

Table 8 – Area with average for data lines for Octaves experiment first trial

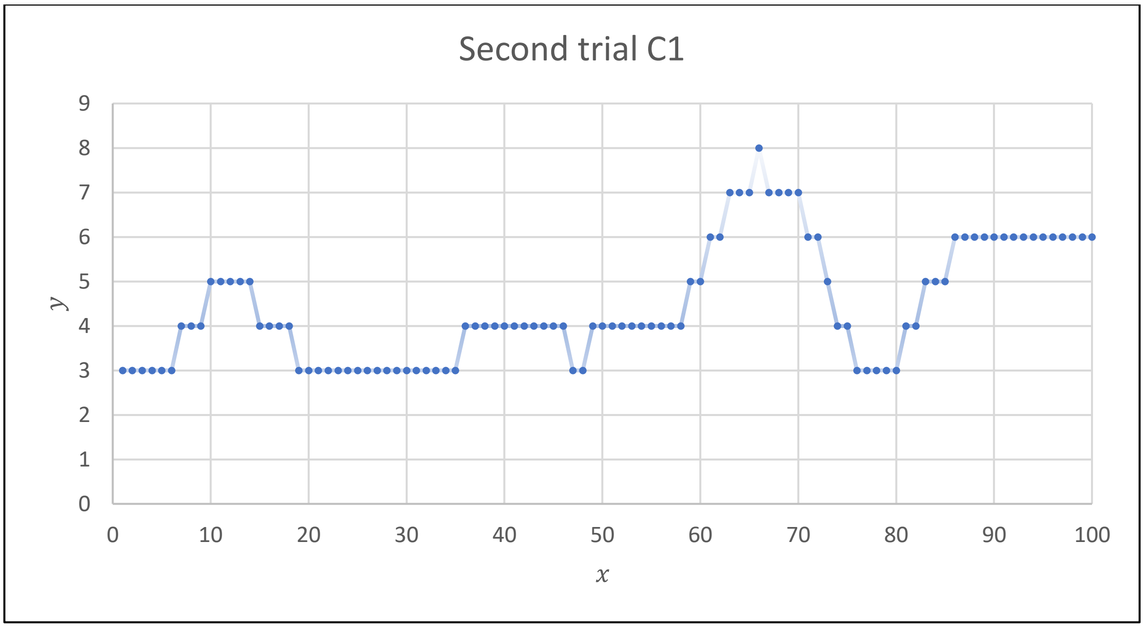
Second trial – Octaves = 4:



Seed: 362
Frequency: 55
Amplitude: 9
Octaves: 4

Seed: 362
Frequency: 55
Amplitude: 9
Octaves: 4

Figure 23 – Terrain for Octaves experiment second trial

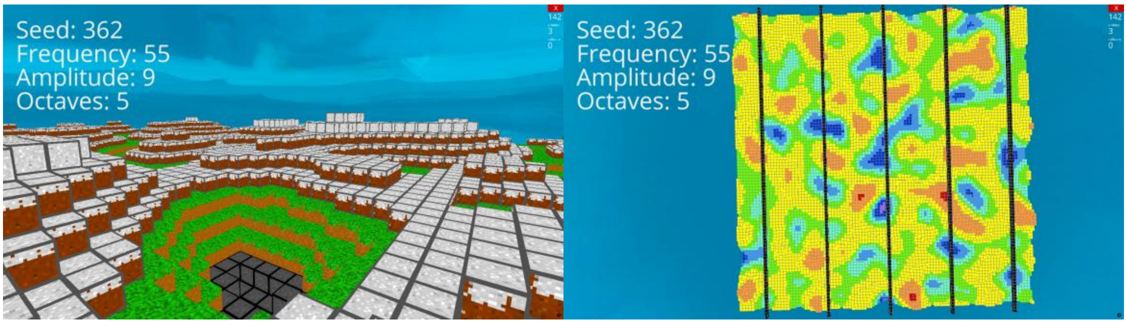


Graph 4 – C1 for Octaves experiment second trial

Data Line	Area
C1	444
C2	460
C3	529
C4	500
C5	497
Average	486

Table 9 – Area with average for data lines for Octaves experiment second trial

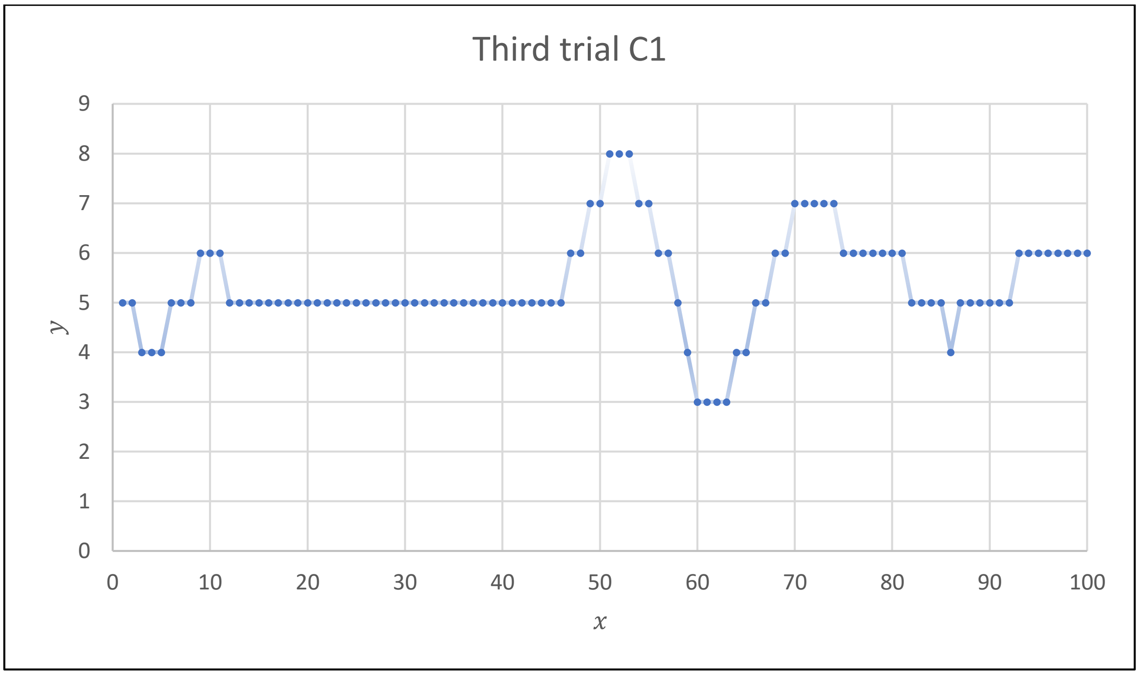
Third trial – Octaves = 5:



Seed: 362
Frequency: 55
Amplitude: 9
Octaves: 5

Seed: 362
Frequency: 55
Amplitude: 9
Octaves: 5

Figure 24 – Terrain for Octaves experiment third trial

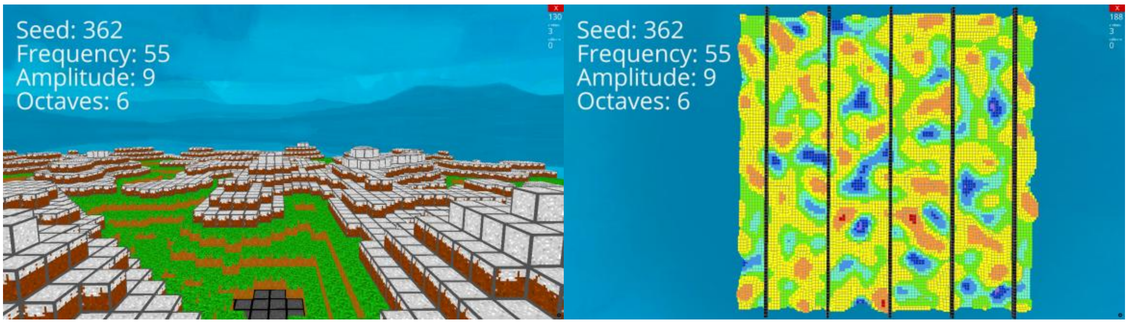


Graph 5 – C1 for Octaves experiment third trial

Data Line	Area
C1	536
C2	561
C3	438
C4	592
C5	485
Average	522

Table 10 – Area with average for data lines for Octaves experiment third trial

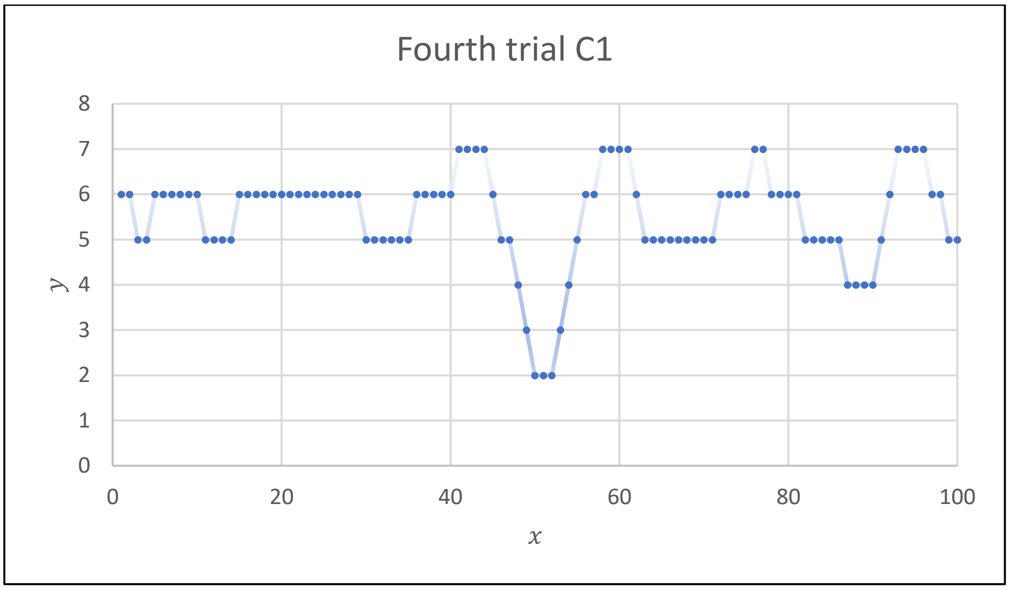
Fourth trial – Octaves = 6:



Seed: 362
Frequency: 55
Amplitude: 9
Octaves: 6

Seed: 362
Frequency: 55
Amplitude: 9
Octaves: 6

Figure 25 – Terrain for Octaves experiment fourth trial



Graph 6 – C1 for Octaves experiment fourth trial

Data Line	Area
C1	552
C2	540
C3	589
C4	601
C5	482
Average	553

Table 11 – Area with average for data lines for Octaves experiment fourth trial

7.4 - Octaves results:

What I can see from *Tables 8-11*, the area has increased after each experiment, octaves 6 giving me the closest number to $Area = 602$ which was Minecraft's terrain area. This means that terrains with 6 octaves are the most similar to Minecraft.

In terms of my hypothesis, in *Figures 22-25* I can see that the variation in colours is increasing. This means that the change in height is more frequent, making the gradient bigger. Hence, when the octaves value is higher, the complexity of the terrain increases, which supports my hypothesis.

8 - Conclusion to my Research Question

For my research I focused on Minecraft as it uses Perlin Noise for terrain generation. Essentially, Minecraft uses various parameters available to create nature-like worlds in the form of blocks, and the simultaneous use of these is what gives the worlds its unique appearance. I identified 3 major parameters during my secondary research that would have the most significant effect on Noise generation, and these are: Amplitude, Frequency and Octaves. My findings during the analysis of the effect of these parameters in my primary research supports the secondary research findings, which is for decreasing frequency, the terrain repeats itself more often, increasing amplitude means the maximum height is increased and lastly, increasing octaves make a terrain more complex.

As I was changing parameters, the visual appearance of terrains changed as described in the Perlin Noise sources I used. Understanding how each parameter works individually and how they work together is needed to successfully generate visually pleasing terrains.

The 4th point of the hypothesis stated that only the analysed parameters influence the noise significantly. My primary research findings state that by using the frequency, amplitude, and octaves a terrain like Minecraft's can be generated. This means these parameters are the most influential, however I noticed that the resulting terrain shapes of the experiments are not exactly the same as the Minecraft terrain I tried to replicate. So, to get the exact Minecraft world generation more parameters must be considered and changed.

9 - Evaluation

The research successfully analysed the individual parameters that affect Minecraft's terrains. Through my experiments, I was able to find visual proof of the noise functions and the effects of its parameters. Thanks to the knowledge gained about the parameters I could then recreate a Minecraft biome while getting insight into how changing parameters change the characteristics of terrains. The terrains generated where coherent and consistent with the traditional Minecraft-like world appearance.

My code allowed me to keep the seed the same, so I could focus on the analysed parameters during experiments. Therefore, I didn't have to worry about the randomicity.

The source I used as reference for the code only explained how to change the frequency, amplitude, octaves, and seed. If I had to repeat the investigation, I would research the exact definition of other parameters as well as a way to change them in my code.

9.1 - Limitations

The conducted experiment has limitations, as my code doesn’t permit a change in every parameter available in Perlin Noise, such as the persistence or “Scale”.[23] This reduces the ability to draw solid conclusions. Being capable of modifying every parameter will help to find a suitable parameter set for each type of terrain.

In addition, using my personal computer system to render the 3D terrain was less efficient than using a specialised PC with dedicated hardware components like the last released version of a GPU. Rendering through a specialised PC, enables analysis of superior width size during the trials, leading to an increased amount of terrain being investigated. Resulting in increased accuracy of the results.

9.2 - Further research

Extra investigation can be done using the full scope of parameters to generate Perlin Noise. As well, exploring how to combine these parameters can be crucial to achieve a seamless flow of ideas throughout the investigation.

Moreover, developing the research in improving optimization strategies and performance to boost the computational efficiency of Perlin Noise in real time generation. Affecting how players see the quality of generating terrain in videogames.

Furthermore, examining how Perlin Noise can be joined with other procedural generation methods like “Wavelet Noise”.[24] This leads to the opportunity of having the benefits from both noises, allowing to utilise the most appropriate in each scenario.

Lastly, Machine learning algorithm could also be a potential integration to Perlin Noise. It may create innovative approaches to generating complex, dynamic and good quality terrain.

[23] (Minetest *Perlin Noise*)

[24] (*Procedural Noise Categories*)

Bibliography

Documents by Ken Perlin - New York University, downloaded from ACM Digital Library

Perlin, Ken. “An image synthesizer.” *ACM SIGGRAPH Computer Graphics*, vol. 19, no. 3, 01 Jul. 1985, pp. 287–296, https://doi.org/10.1145/325165.325247. Accessed 9 Nov. 2023.

Perlin, Ken. “Improving noise.” *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, 01, Jul. 2002, https://doi.org/10.1145/566570.566636. Accessed 9 Nov. 2023.

Videogame Minecraft:

Microsoft. *Minecraft*. Version 1.16.5 Java Edition, Mojang, 2021. Accessed 9 Nov. 2023.

Videos for code:

Red Hen Dev, director. *Python Minecraft with Ursina: Biomes with Texture Atlas, Terrain from Mesh - Part 1 (Part 22)*. *YouTube*, YouTube, 14 Nov. 2021, https://www.youtube.com/watch?v=fofR4yzTCdA. Accessed 9 Nov. 2023.

Red Hen Dev, director. *Python Minecraft with Ursina: Procedural Perlin Noise Terrain and Simple Biomes - Part 2*. *YouTube*, YouTube, 18 Nov. 2021, https://www.youtube.com/watch?v=yAMNyWcuozA&t=2s. Accessed 9 Nov. 2023.

Websites:

Bevins, Jason. “Tutorial 4: Modifying the Parameters of the Noise Module.” *Libnoise*, 2005, libnoise.sourceforge.net/tutorials/tutorial4.html. Accessed 9 Nov. 2023.

Biagioli, Adrian. “Adrian’s Soapbox.” *Understanding Perlin Noise*, 9 Aug. 2014, adrianb.io/2014/08/09/perlinnoise.html. Accessed 9 Nov. 2023.

Compton, Caleb. “How Minecraft Generates Massive Virtual Worlds from Scratch.” *How Minecraft Generates Massive Virtual Worlds from Scratch*, 28 Feb. 2021, remptongames.com/2021/02/28/how-minecraft-generates-massive-virtual-worlds-from-scratch/. Accessed 9 Nov. 2023.

Elias, Hugo. “Perlin Noise.” *Wayback Machine*, web.archive.org/web/20160510013854/freespace.virgin.net/hugo.elias/models/m_perlin.htm. Accessed 26 Feb. 2024.

Fingas, Jon. “Here’s How ‘Minecraft’ Creates Its Gigantic Worlds.” *Engadget*, Engadget, 19 July 2019, www.engadget.com/2015-03-04-how-minecraft-worlds-are-made.html. Accessed 9 Nov. 2023.

Hirnschall, Sebastian. *Perlin Noise: What Is It, and How to Use It.*, 27 Aug. 2020, blog.hirnschall.net/perlin-noise/. Accessed 9 Nov. 2023.

Minecraft Wiki. “Terrain Features.” *Minecraft Wiki*, Fandom, Inc., 23 Sept. 2023, minecraft.fandom.com/wiki/Terrain_features. Accessed 9 Nov. 2023.

Minetest. “Perlin Noise.” *Perlin Noise - Minetest API Documentation*, minetest.gitlab.io/minetest/perlin-noise/. Accessed 9 Nov. 2023.

Perlin, Ken. “Noise and Turbulence.” *Ken’s Academy Award*, cs.nyu.edu/~perlin/doc/oscar.html. Accessed 26 Feb. 2024.

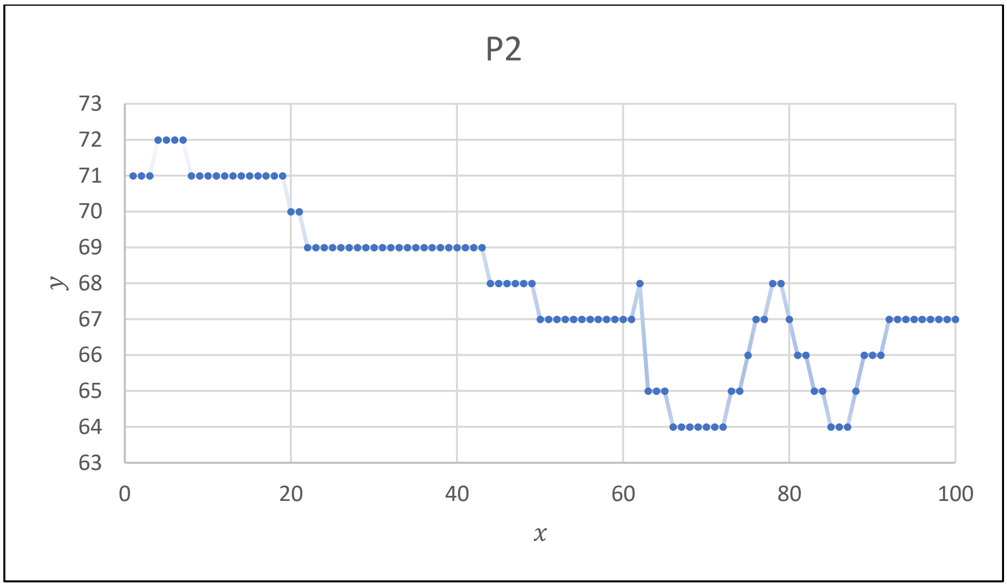
“Procedural Noise Categories.” *PhysBAM*, 29 Mar. 2011, physbam.stanford.edu/cs448x/old/Procedural_Noise(2f)Categories.html. Accessed 26 Feb. 2024.

Red Hen Dev. “perlin_module.py.” *GitHub*, 23 Feb. 2022, github.com/RedHenDev/ursina_tutorials/blob/main/python_meshCraft_tut_2021/perlin_module.py. Accessed 9 Nov. 2023.

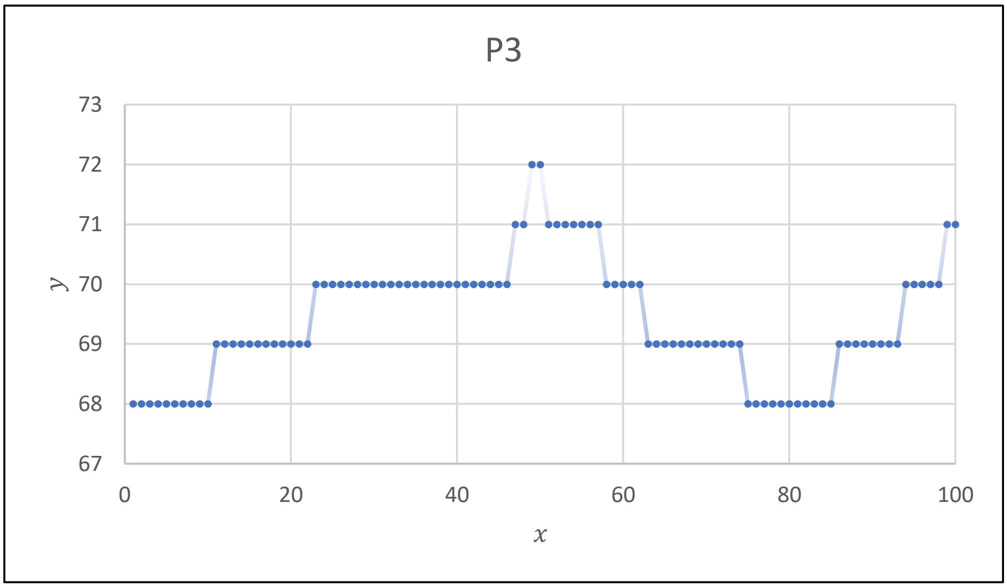
“Understanding Some Basic Noise Terms.” *Extension Pack for MARI*, campi3d.com/External/MariExtensionPack/userGuide5R8/Understandingsomebasicnoiseterms.html. Accessed 26 Feb. 2024.

Appendix

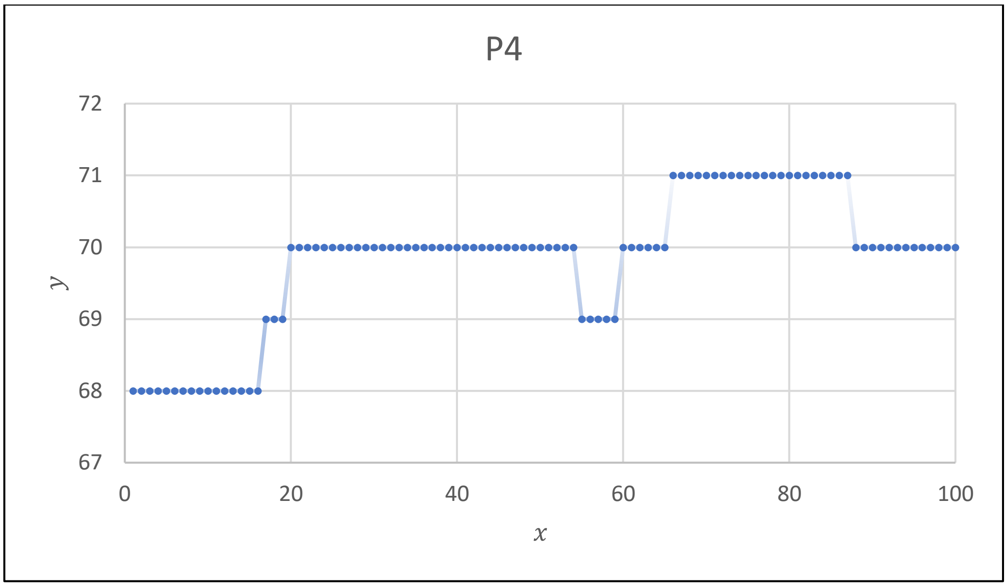
Graphs from primary research:



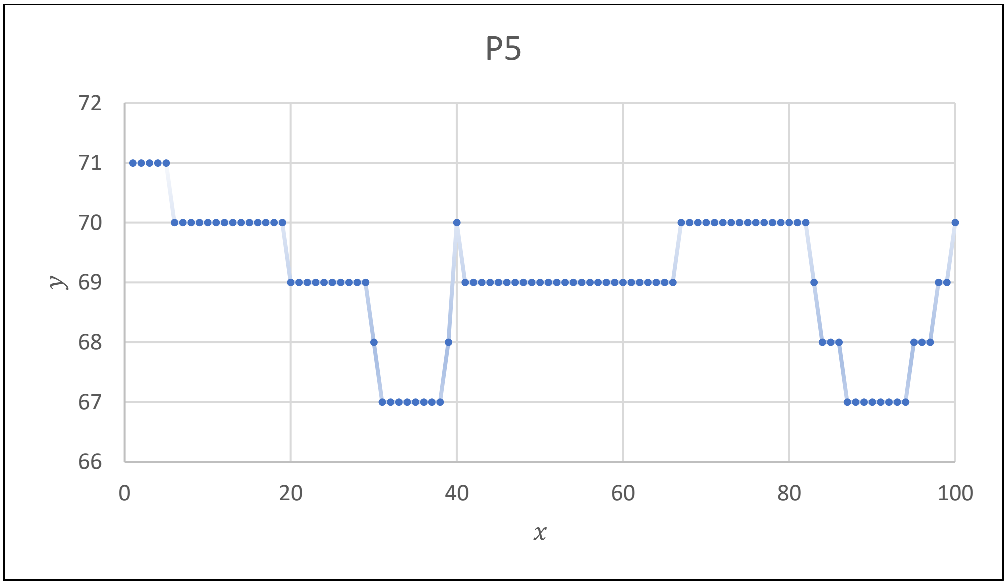
Graph 7 – P2 data for block position



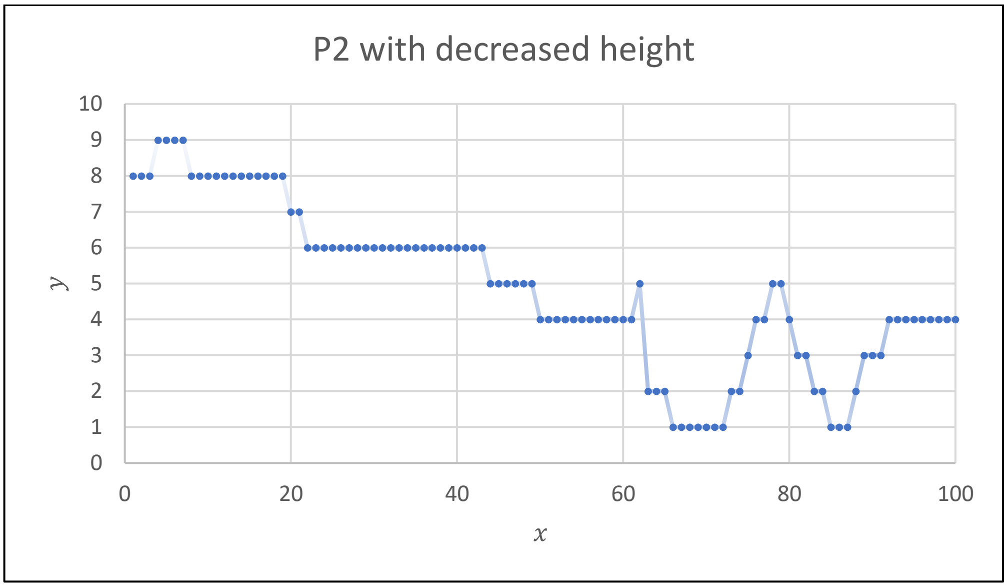
Graph 8 – P3 data for block position



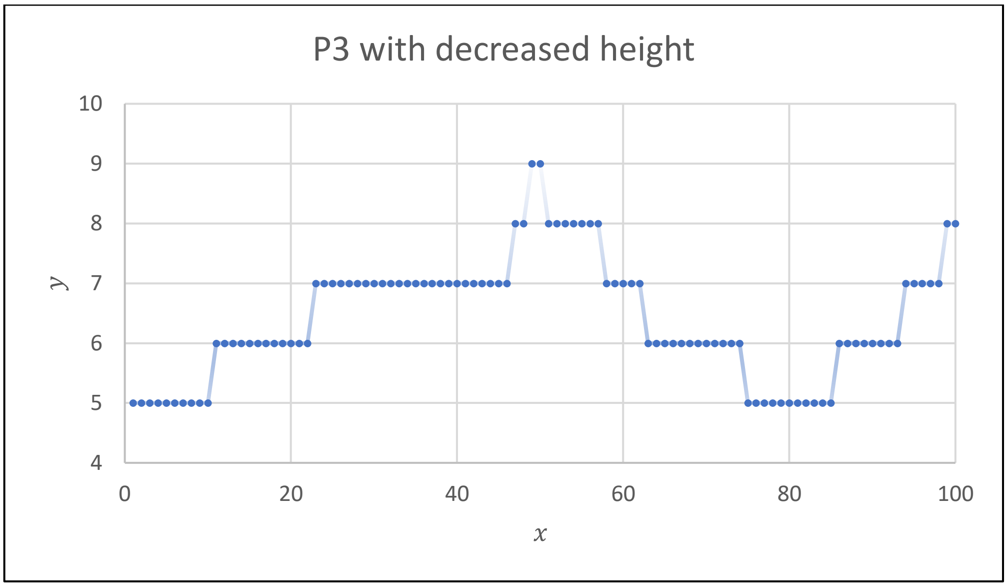
Graph 9 – P4 data for block position



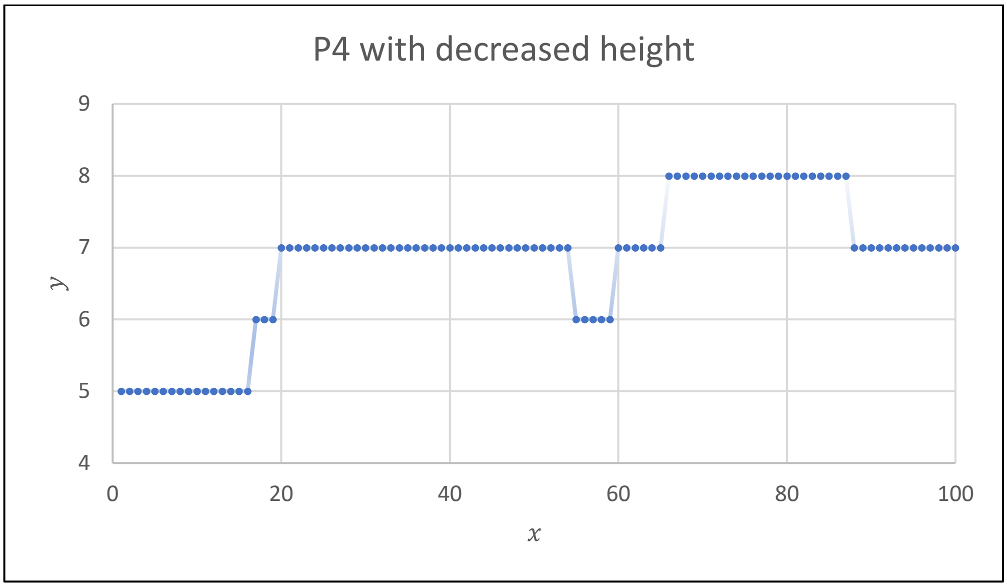
Graph 10 – P5 data for block position



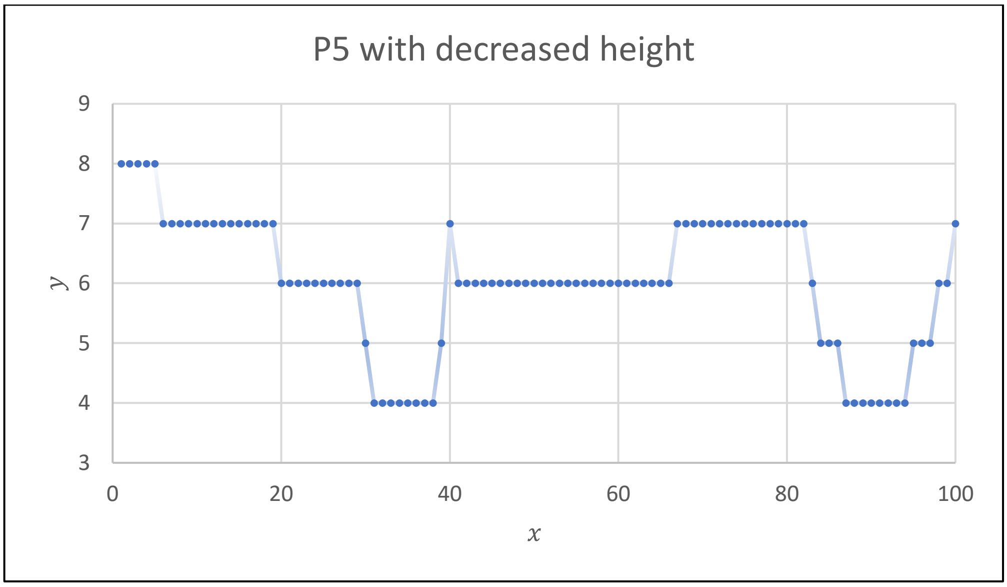
Graph 11 – P2 data for block position with decreased height



Graph 12 – P3 data for block position with decreased height

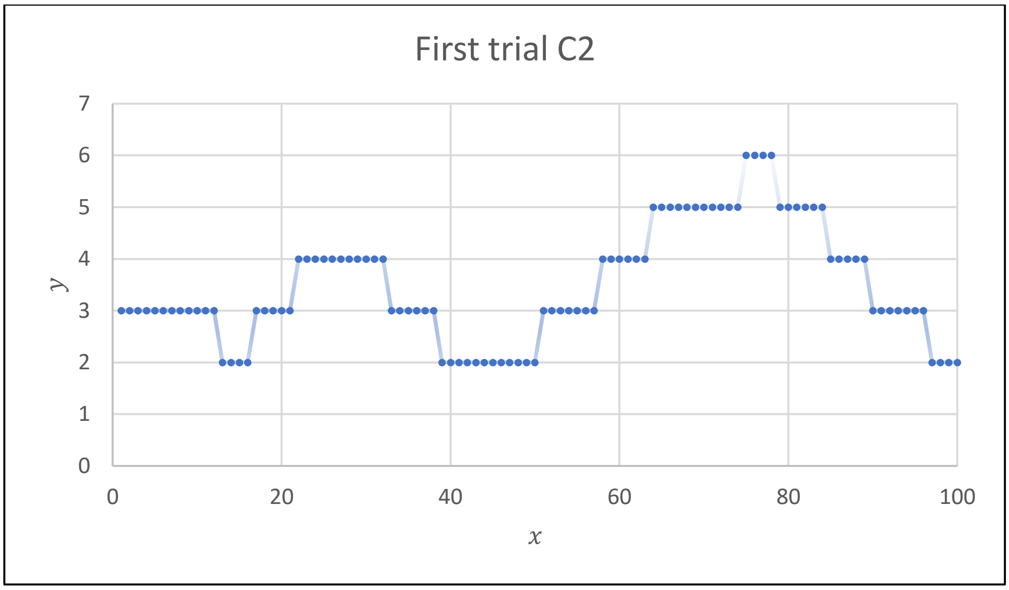


Graph 13 – P4 data for block position with decreased height

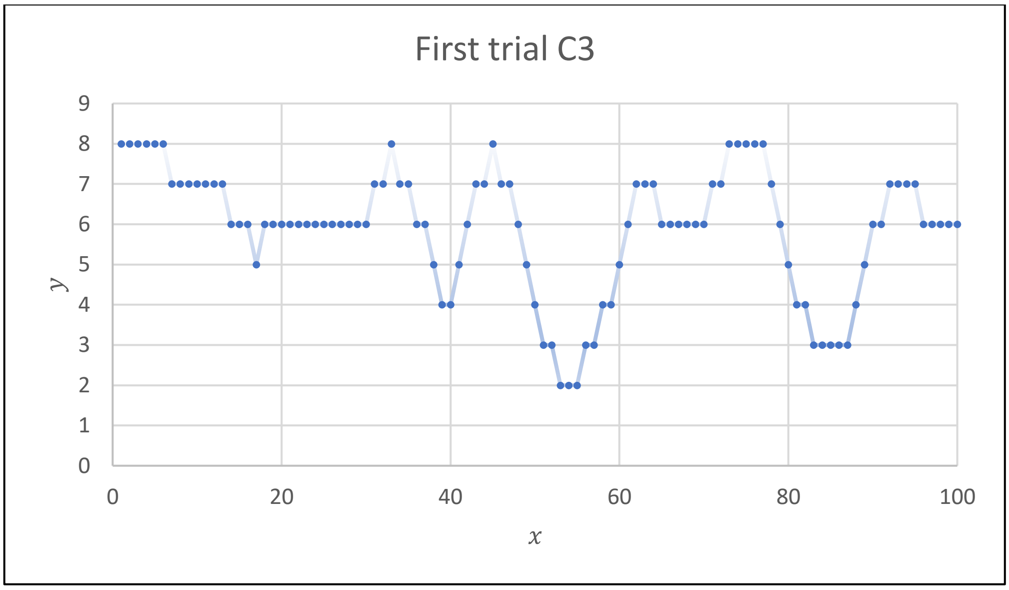


Graph 14 – P5 data for block position with decreased height

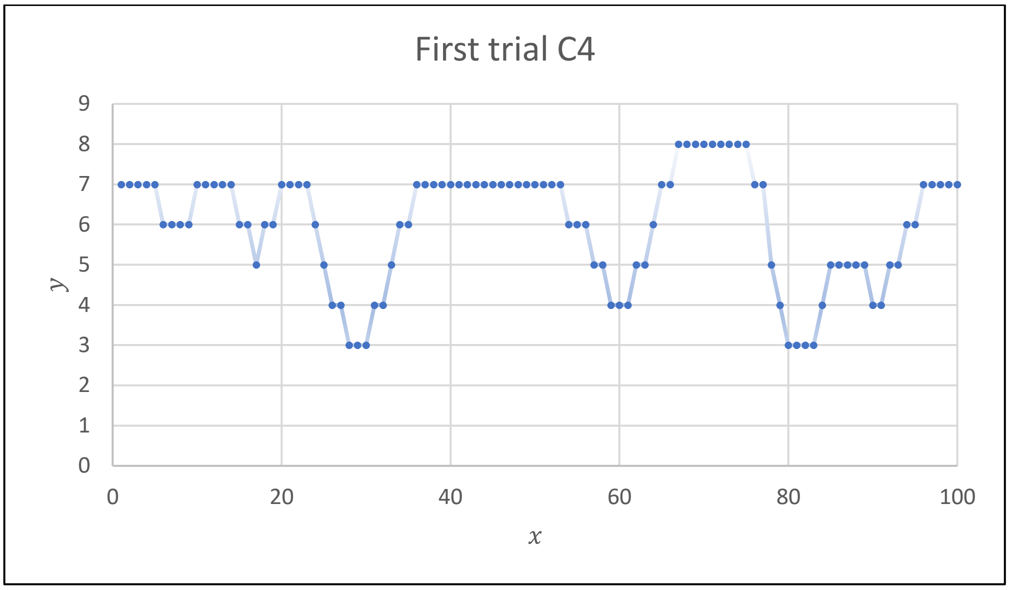
Graphs for Octaves experiment:



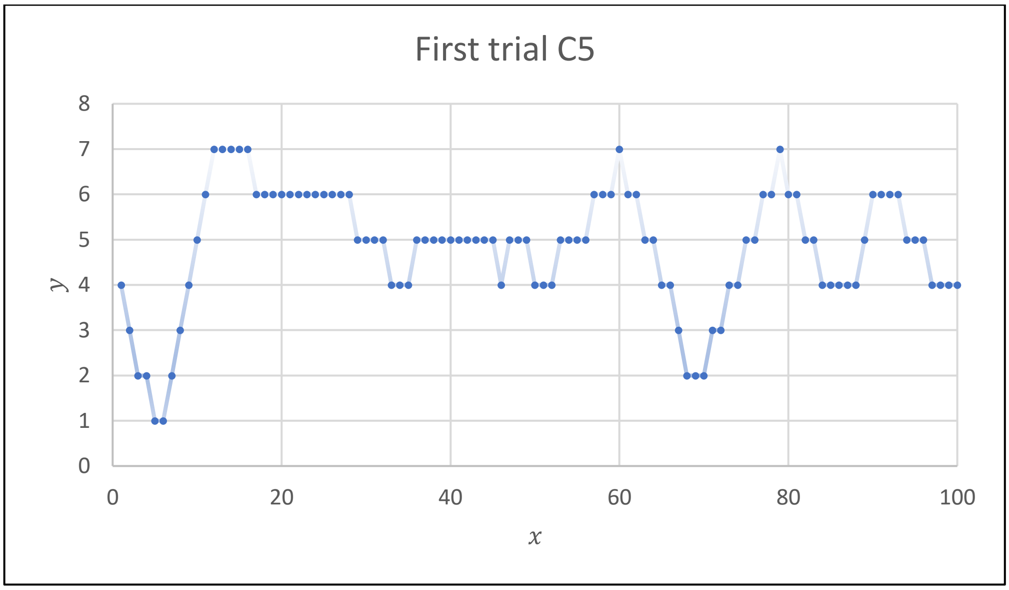
Graph 15 – C2 for Octaves experiment first trial



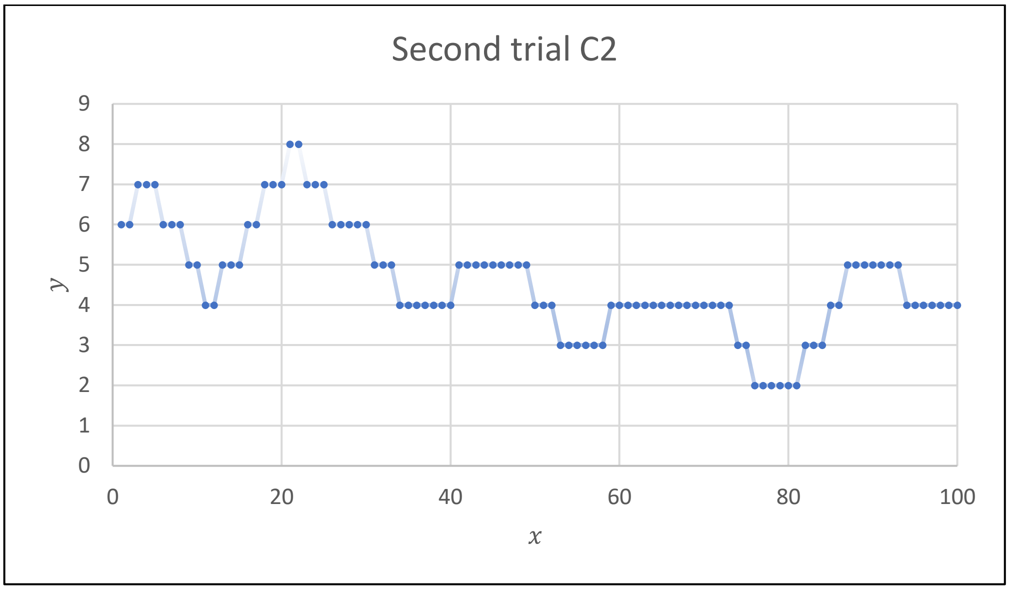
Graph 16 – C3 for Octaves experiment first trial



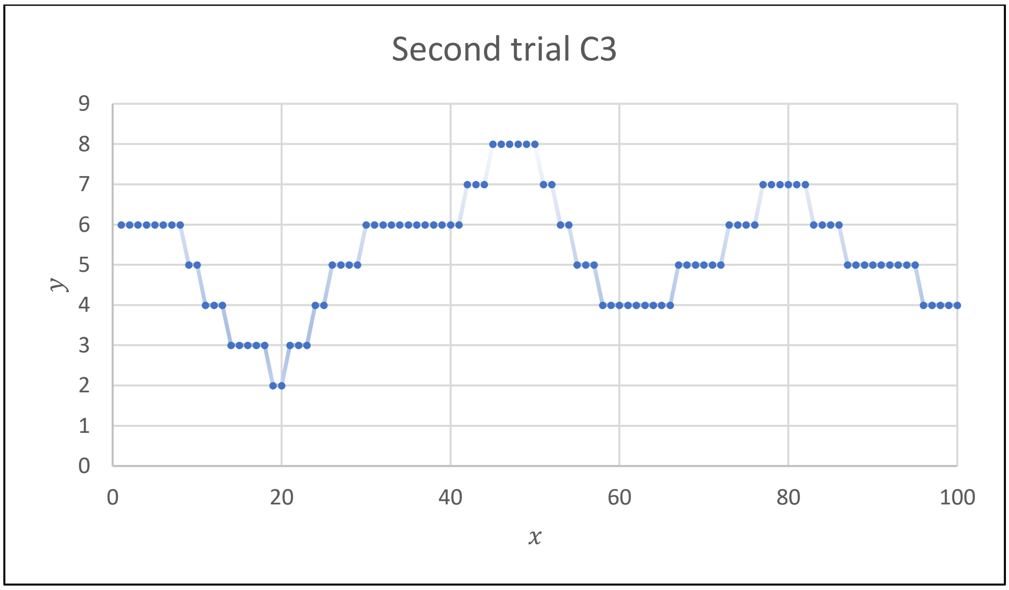
Graph 17 – C4 for Octaves experiment first trial



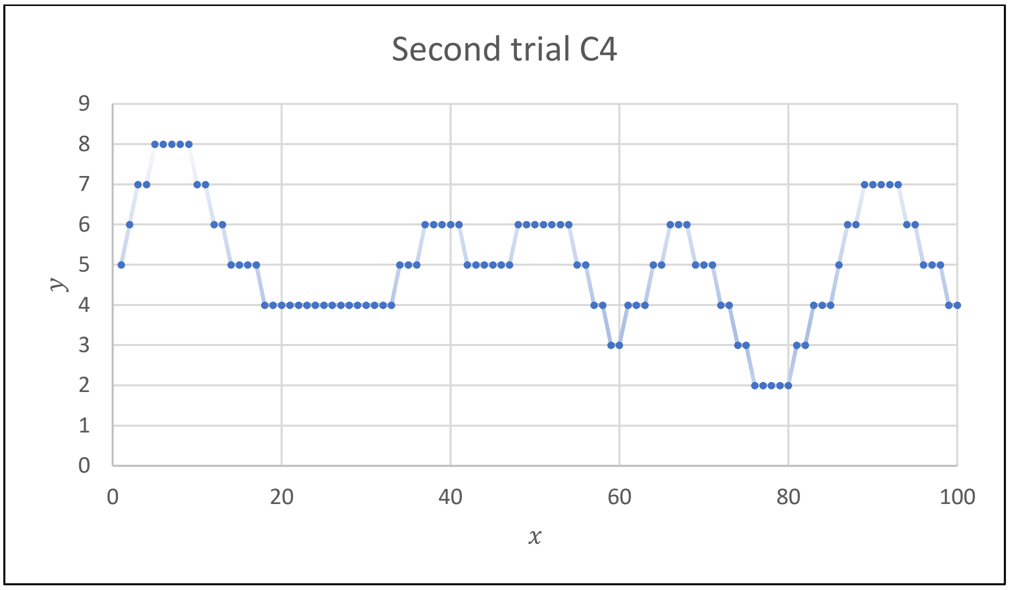
Graph 18 – C5 for Octaves experiment first trial



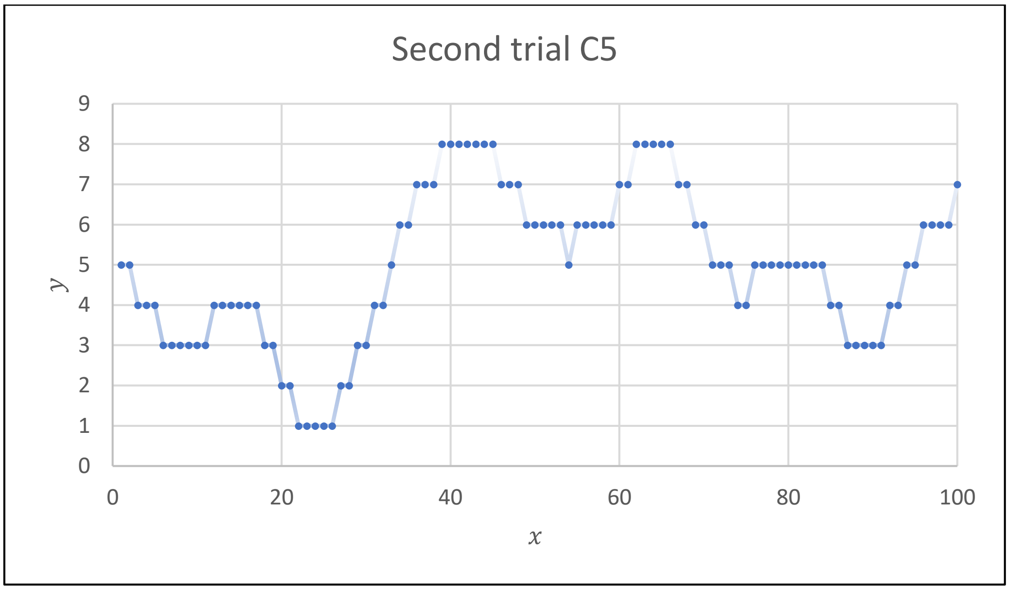
Graph 19 – C2 for Octaves experiment second trial



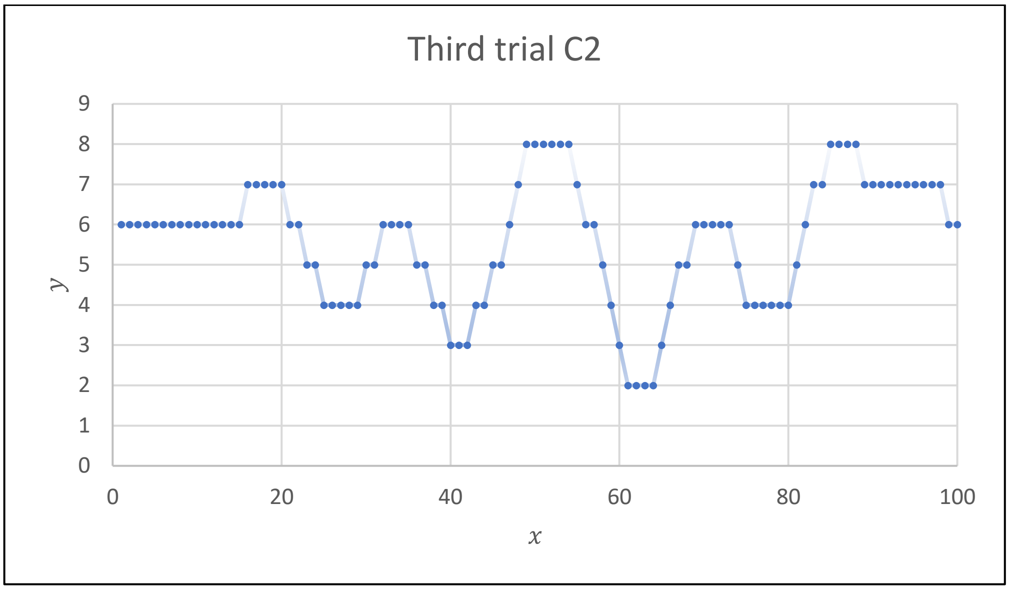
Graph 20 – C3 for Octaves experiment second trial



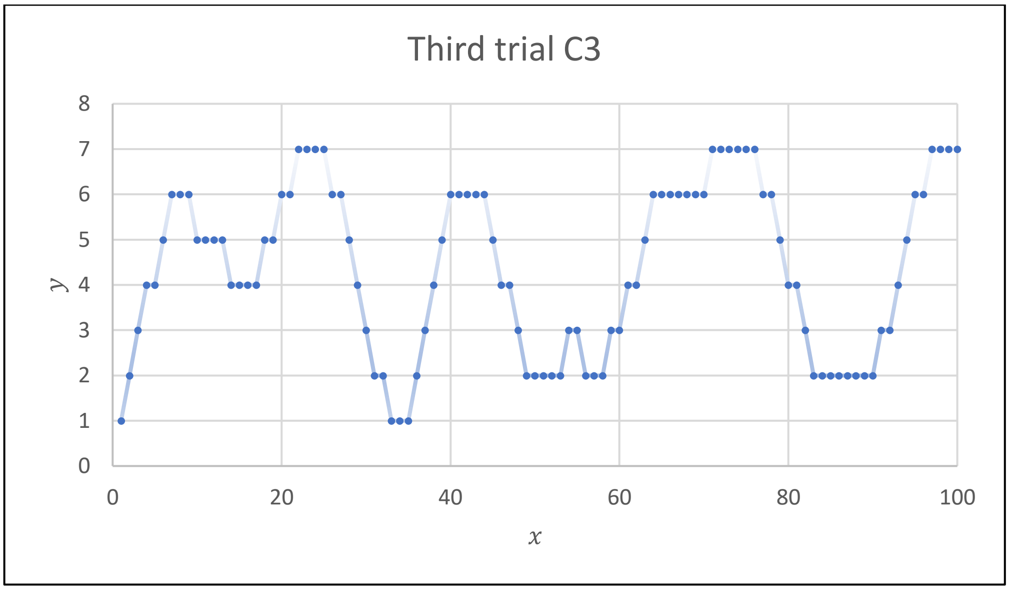
Graph 21 – C4 for Octaves experiment second trial



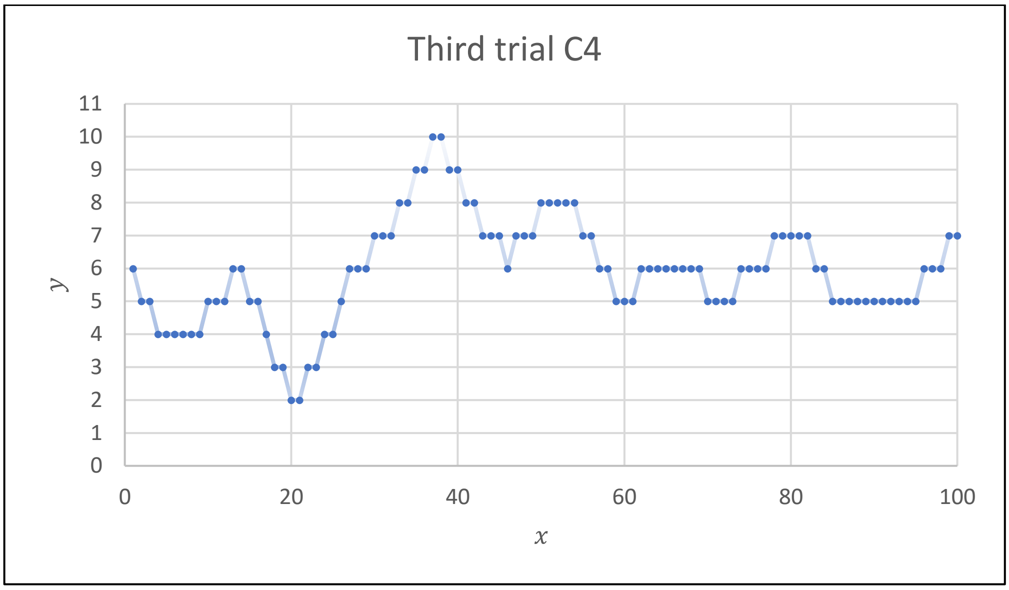
Graph 22 – C5 for Octaves experiment second trial



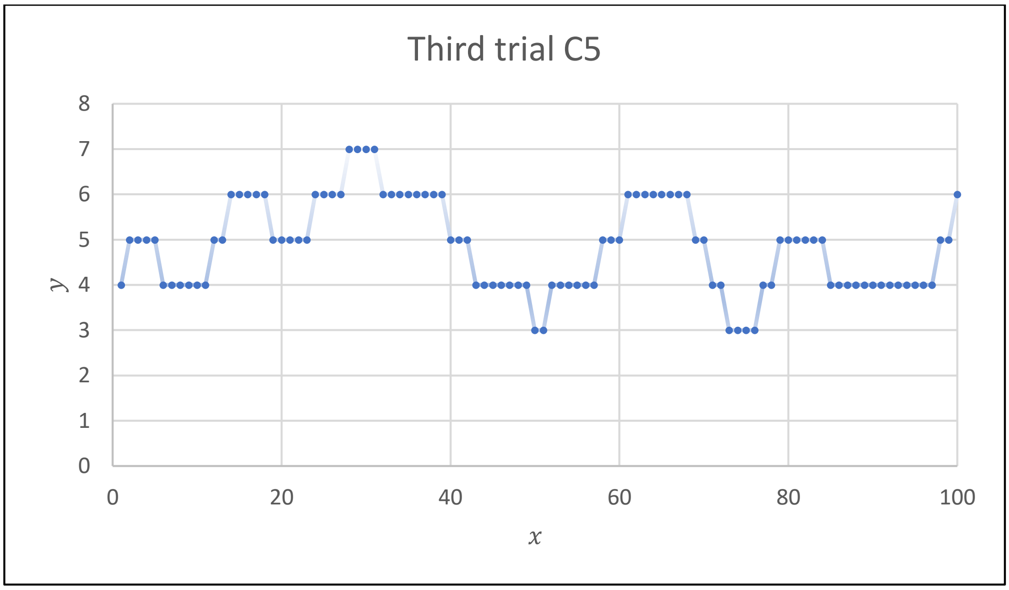
Graph 23 – C2 for Octaves experiment third trial



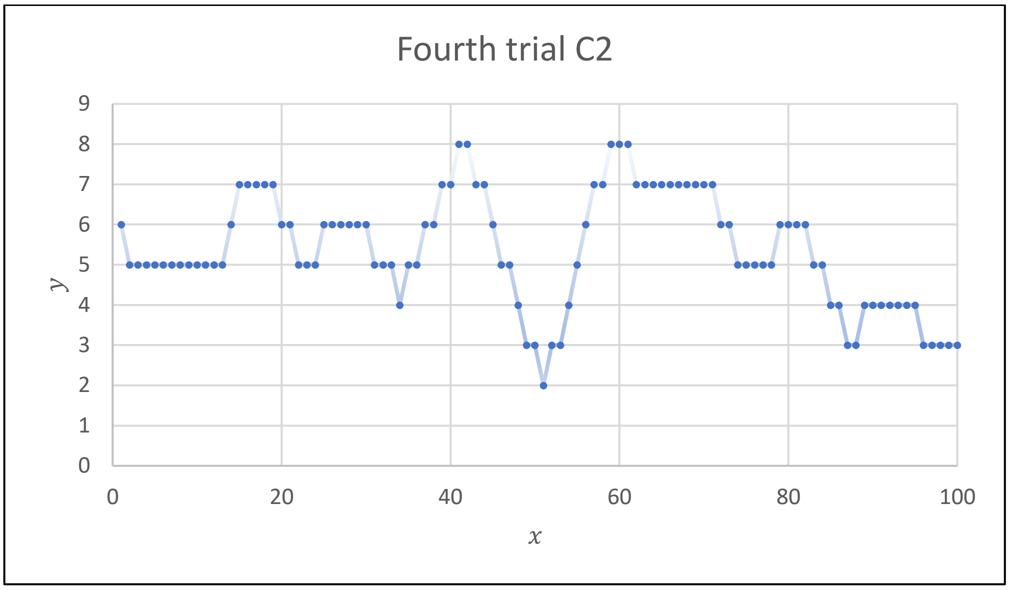
Graph 24 – C3 for Octaves experiment third trial



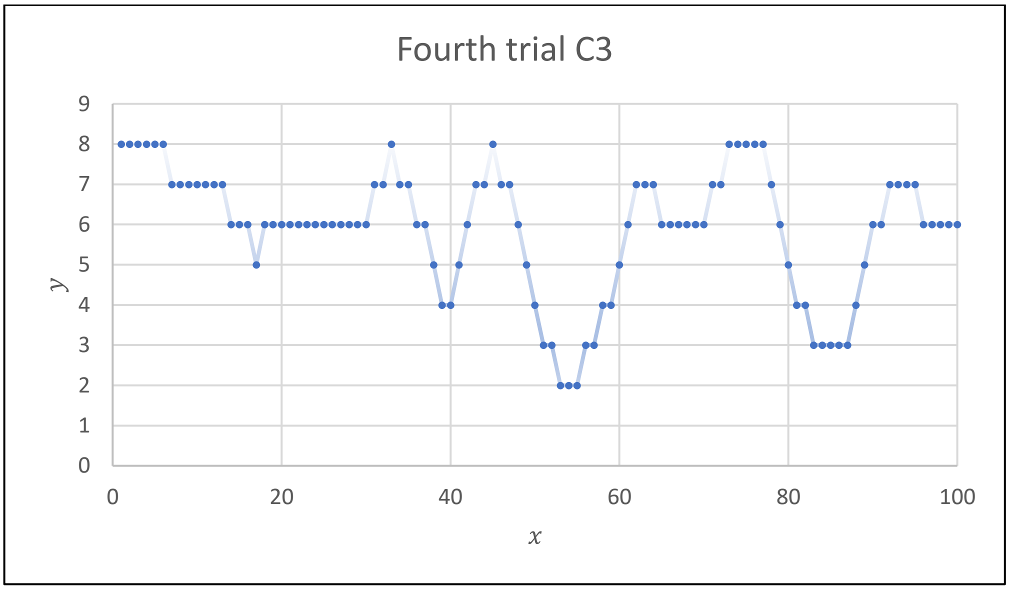
Graph 25 – C4 for Octaves experiment third trial



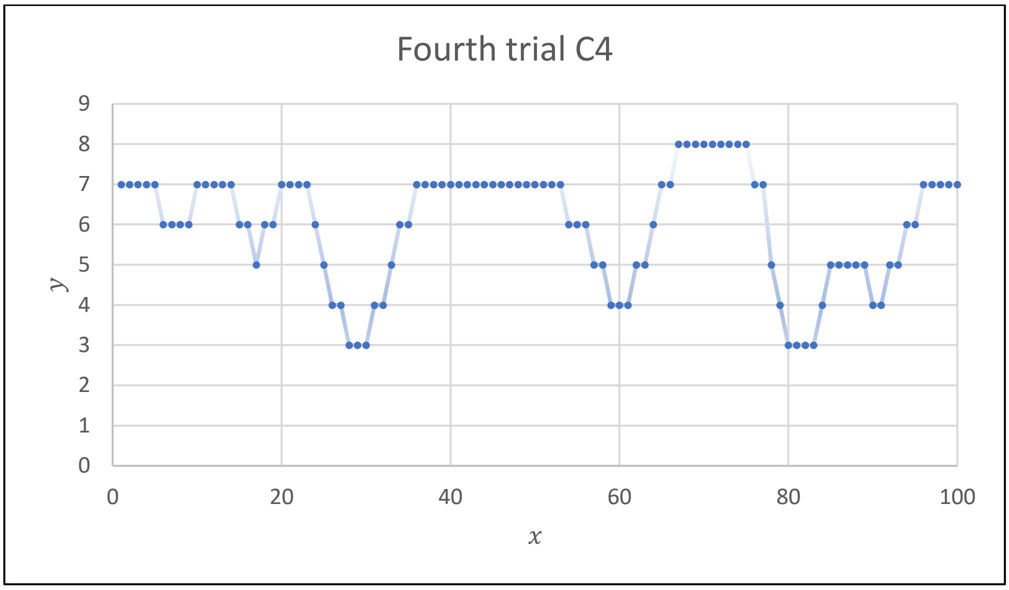
Graph 26 – C5 for Octaves experiment third trial



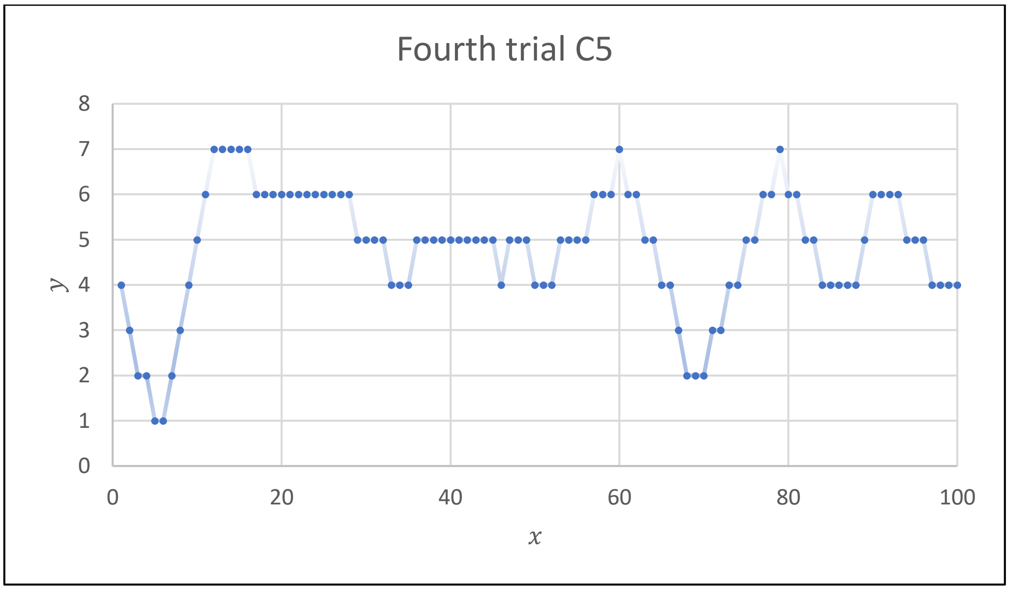
Graph 27 – C2 for Octaves experiment fourth trial



Graph 28 – C3 for Octaves experiment fourth trial



Graph 29 – C4 for Octaves experiment fourth trial



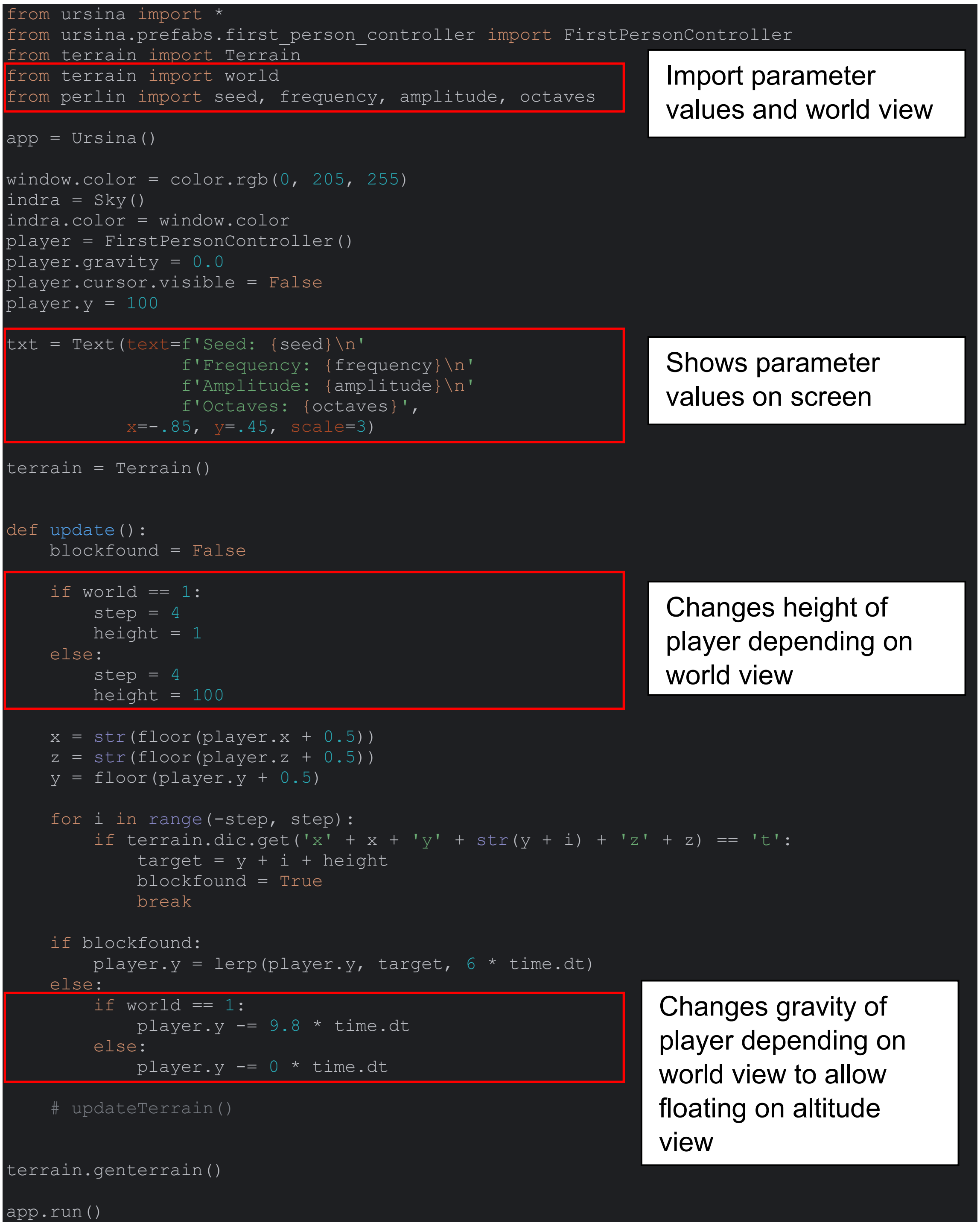
Graph 30 – C5 for Octaves experiment fourth trial

Program for primary research experiment:

Lines I wrote or edited are in red boxes. Function of these lines are in text boxes.

Other lines from YouTube tutorials.

main.py



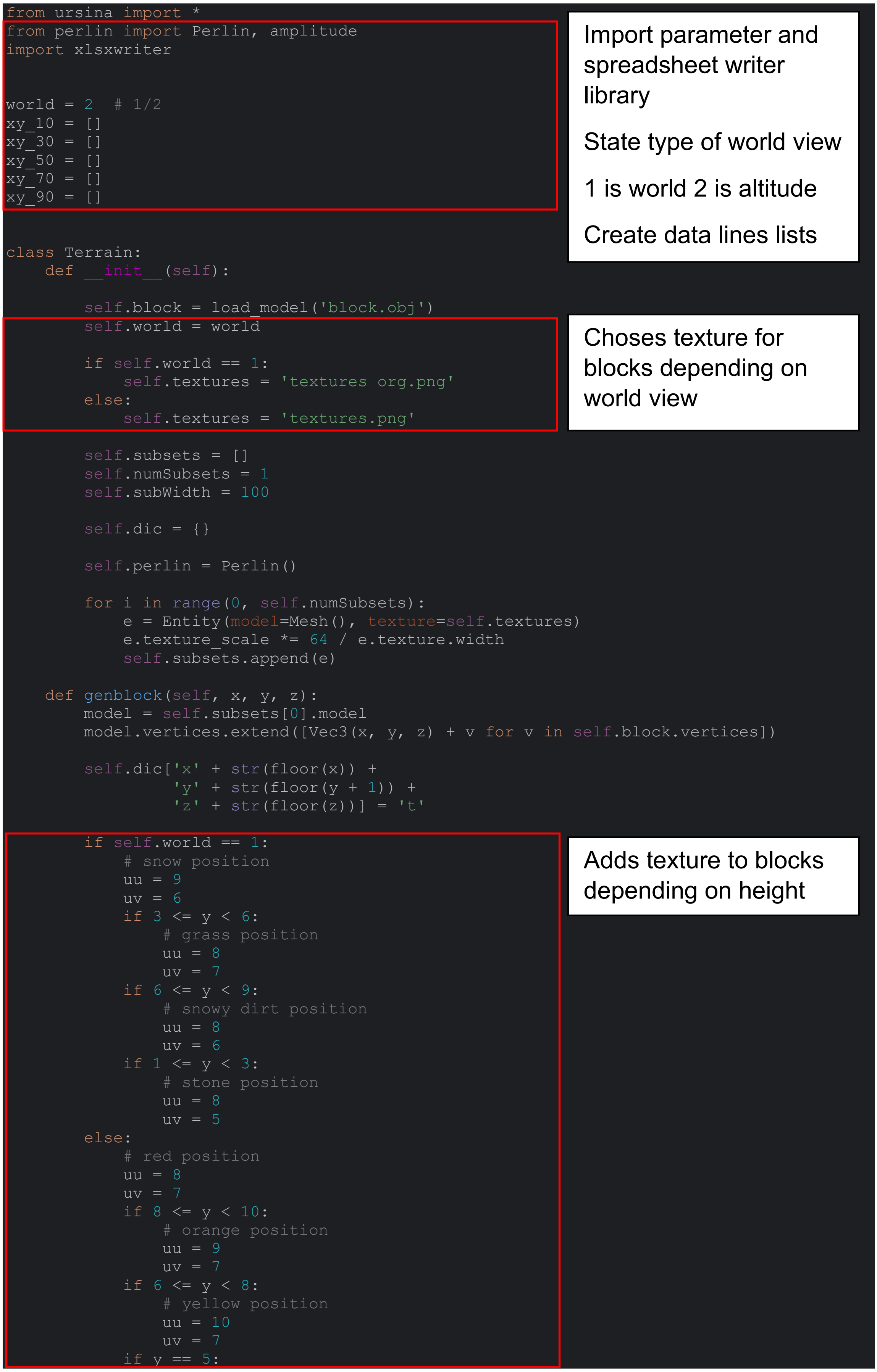
```
from ursina import *
from ursina.prefabs.first_person_controller import FirstPersonController
from terrain import Terrain
from terrain import world
from perlin import seed, frequency, amplitude, octaves

app = Ursina()

window.color = color.rgb(0, 205, 255)
indra = Sky()
indra.color = window.color
player = FirstPersonController()
player.gravity = 0.0
player.cursor.visible = False
player.y = 100

txt = Text(text=f'Seed: {seed}\n'
                f'Frequency: {frequency}\n'
                f'Amplitude: {amplitude}\n'
                f'Octaves: {octaves}',
           x=-.85, y=.45, scale=3)

terrain = Terrain()


def update():
    blockfound = False

    if world == 1:
        step = 4
        height = 1
    else:
        step = 4
        height = 100

    x = str(floor(player.x + 0.5))
    z = str(floor(player.z + 0.5))
    y = floor(player.y + 0.5)

    for i in range(-step, step):
        if terrain.dic.get('x' + x + 'y' + str(y + i) + 'z' + z) == 't':
            target = y + i + height
            blockfound = True
            break

    if blockfound:
        player.y = lerp(player.y, target, 6 * time.dt)
    else:
        if world == 1:
            player.y -= 9.8 * time.dt
        else:
            player.y -= 0 * time.dt

    # updateTerrain()


terrain.genterrain()

app.run()
```

Import parameter values and world view

Shows parameter values on screen

Changes height of player depending on world view

Changes gravity of player depending on world view to allow floating on altitude view terrain.py



```
from ursina import *
from perlin import Perlin, amplitude
import xlsxwriter


world = 2  # 1/2
xy_10 = []
xy_30 = []
xy_50 = []
xy_70 = []
xy_90 = []


class Terrain:
    def __init__(self):

        self.block = load_model('block.obj')
        self.world = world

        if self.world == 1:
            self.textures = 'textures org.png'
        else:
            self.textures = 'textures.png'

        self.subsets = []
        self.numSubsets = 1
        self.subWidth = 100

        self.dic = {}

        self.perlin = Perlin()

        for i in range(0, self.numSubsets):
            e = Entity(model=Mesh(), texture=self.textures)
            e.texture_scale *= 64 / e.texture.width
            self.subsets.append(e)

    def genblock(self, x, y, z):
        model = self.subsets[0].model
        model.vertices.extend([Vec3(x, y, z) + v for v in self.block.vertices])

        self.dic['x' + str(floor(x)) +
                 'y' + str(floor(y + 1)) +
                 'z' + str(floor(z))] = 't'

        if self.world == 1:
            # snow position
            uu = 9
            uv = 6
            if 3 <= y < 6:
                # grass position
                uu = 8
                uv = 7
            if 6 <= y < 9:
                # snowy dirt position
                uu = 8
                uv = 6
            if 1 <= y < 3:
                # stone position
                uu = 8
                uv = 5
        else:
            # red position
            uu = 8
            uv = 7
            if 8 <= y < 10:
                # orange position
                uu = 9
                uv = 7
            if 6 <= y < 8:
                # yellow position
                uu = 10
                uv = 7
            if y == 5:
```

Import parameter and spreadsheet writer library

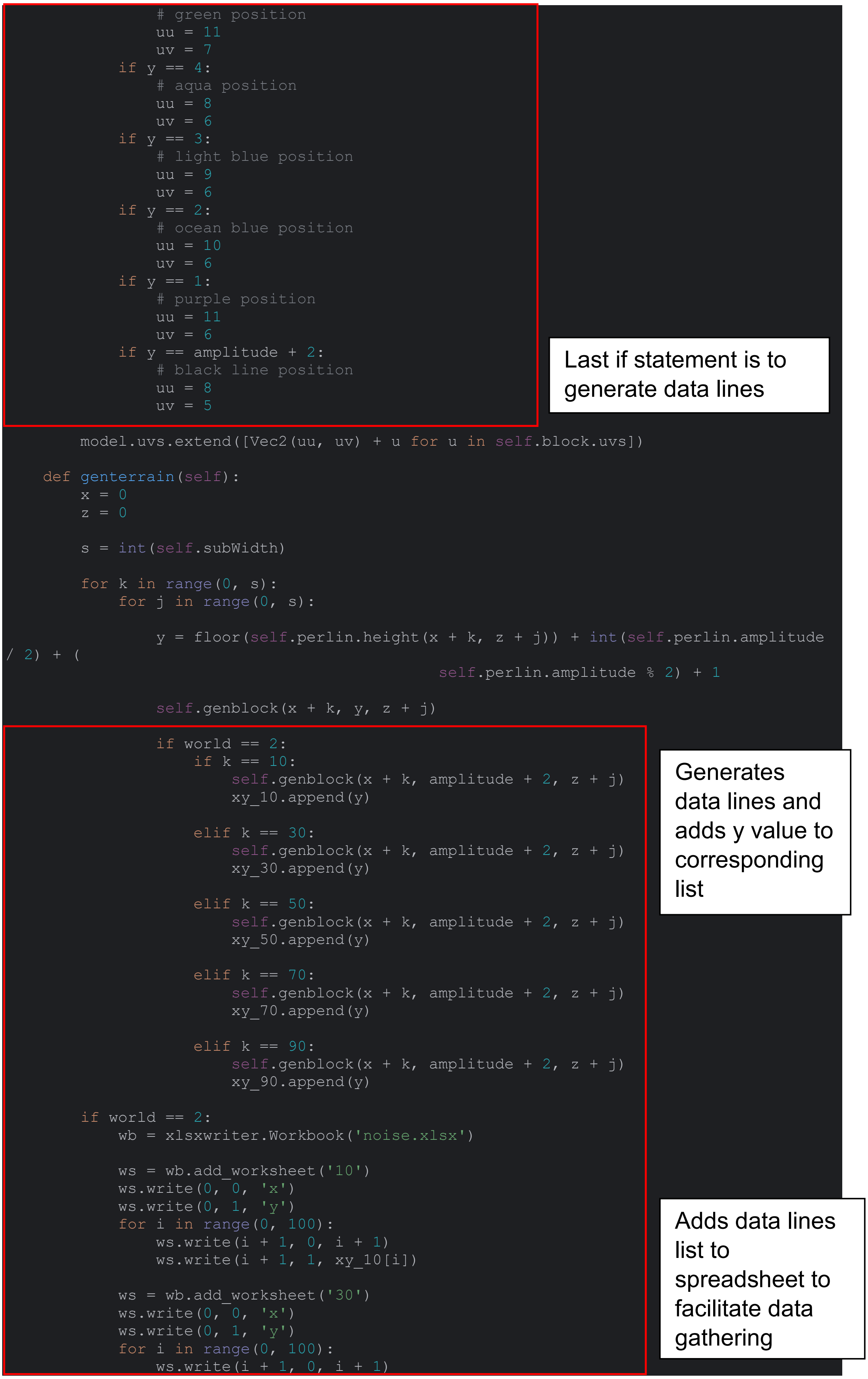
State type of world view

1 is world 2 is altitude

Create data lines lists

Choses texture for blocks depending on world view

Adds texture to blocks depending on height

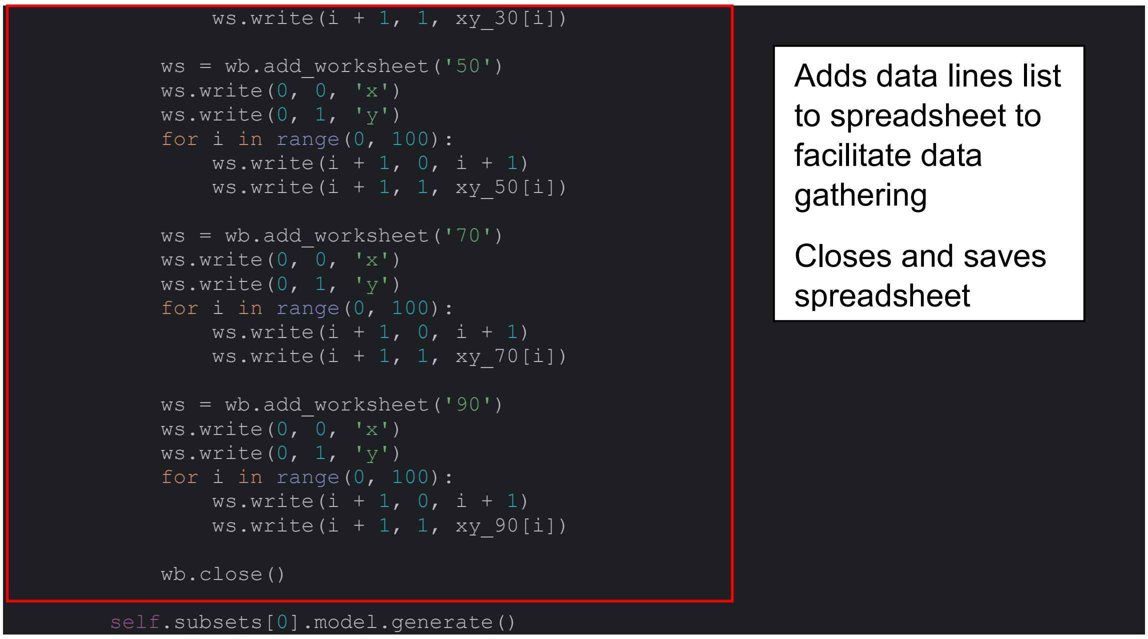


```
                # green position
                uu = 11
                uv = 7
            if y == 4:
                # aqua position
                uu = 8
                uv = 6
            if y == 3:
                # light blue position
                uu = 9
                uv = 6
            if y == 2:
                # ocean blue position
                uu = 10
                uv = 6
            if y == 1:
                # purple position
                uu = 11
                uv = 6
            if y == amplitude + 2:
                # black line position
                uu = 8
                uv = 5
        model.uvs.extend([Vec2(uu, uv) + u for u in self.block.uvs])

    def genterrain(self):
        x = 0
        z = 0

        s = int(self.subWidth)

        for k in range(0, s):
            for j in range(0, s):

                y = floor(self.perlin.height(x + k, z + j)) + int(self.perlin.amplitude
/ 2) + (
                                              self.perlin.amplitude % 2) + 1

                self.genblock(x + k, y, z + j)

                if world == 2:
                    if k == 10:
                        self.genblock(x + k, amplitude + 2, z + j)
                        xy_10.append(y)

                    elif k == 30:
                        self.genblock(x + k, amplitude + 2, z + j)
                        xy_30.append(y)

                    elif k == 50:
                        self.genblock(x + k, amplitude + 2, z + j)
                        xy_50.append(y)

                    elif k == 70:
                        self.genblock(x + k, amplitude + 2, z + j)
                        xy_70.append(y)

                    elif k == 90:
                        self.genblock(x + k, amplitude + 2, z + j)
                        xy_90.append(y)

        if world == 2:
            wb = xlsxwriter.Workbook('noise.xlsx')

            ws = wb.add_worksheet('10')
            ws.write(0, 0, 'x')
            ws.write(0, 1, 'y')
            for i in range(0, 100):
                ws.write(i + 1, 0, i + 1)
                ws.write(i + 1, 1, xy_10[i])

            ws = wb.add_worksheet('30')
            ws.write(0, 0, 'x')
            ws.write(0, 1, 'y')
            for i in range(0, 100):
                ws.write(i + 1, 0, i + 1)
```

Last if statement is to generate data lines

Generates data lines and adds y value to corresponding list

Adds data lines list to spreadsheet to facilitate data gathering

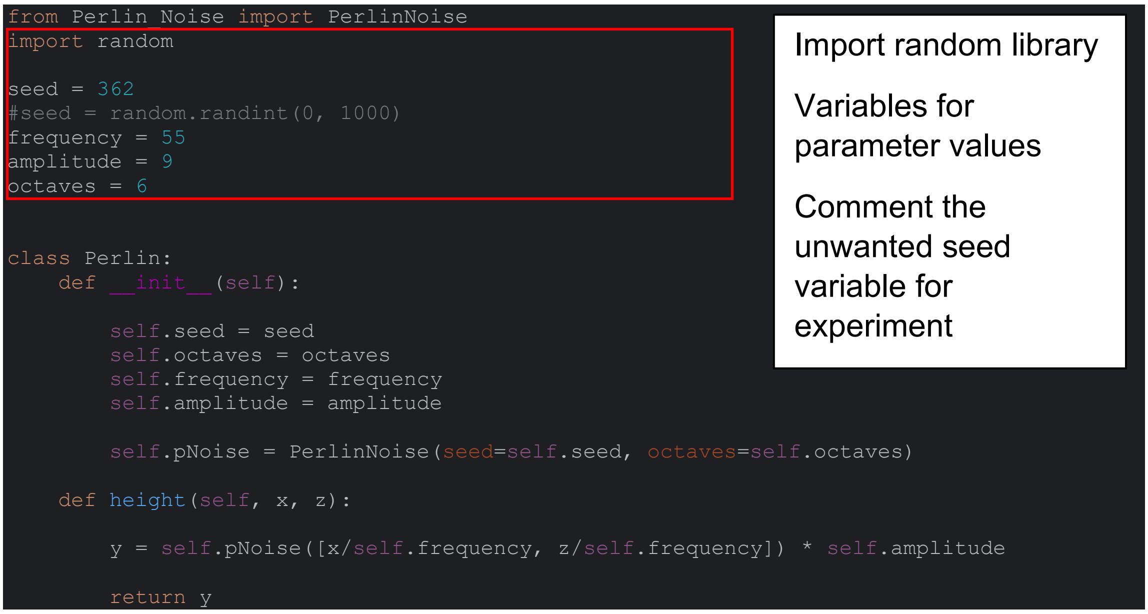


```
                ws.write(i + 1, 1, xy_30[i])

            ws = wb.add_worksheet('50')
            ws.write(0, 0, 'x')
            ws.write(0, 1, 'y')
            for i in range(0, 100):
                ws.write(i + 1, 0, i + 1)
                ws.write(i + 1, 1, xy_50[i])

            ws = wb.add_worksheet('70')
            ws.write(0, 0, 'x')
            ws.write(0, 1, 'y')
            for i in range(0, 100):
                ws.write(i + 1, 0, i + 1)
                ws.write(i + 1, 1, xy_70[i])

            ws = wb.add_worksheet('90')
            ws.write(0, 0, 'x')
            ws.write(0, 1, 'y')
            for i in range(0, 100):
                ws.write(i + 1, 0, i + 1)
                ws.write(i + 1, 1, xy_90[i])

            wb.close()
        self.subsets[0].model.generate()
```

Adds data lines list to spreadsheet to facilitate data gathering

Closes and saves spreadsheet

perlin.py



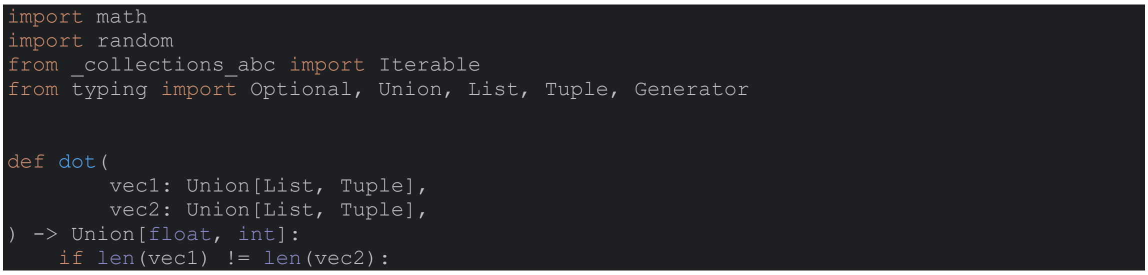
```
from Perlin_Noise import PerlinNoise
import random

seed = 362
#seed = random.randint(0, 1000)
frequency = 55
amplitude = 9
octaves = 6


class Perlin:
    def __init__(self):

        self.seed = seed
        self.octaves = octaves
        self.frequency = frequency
        self.amplitude = amplitude

        self.pNoise = PerlinNoise(seed=self.seed, octaves=self.octaves)

    def height(self, x, z):

        y = self.pNoise([x/self.frequency, z/self.frequency]) * self.amplitude

        return y
```

Import random library

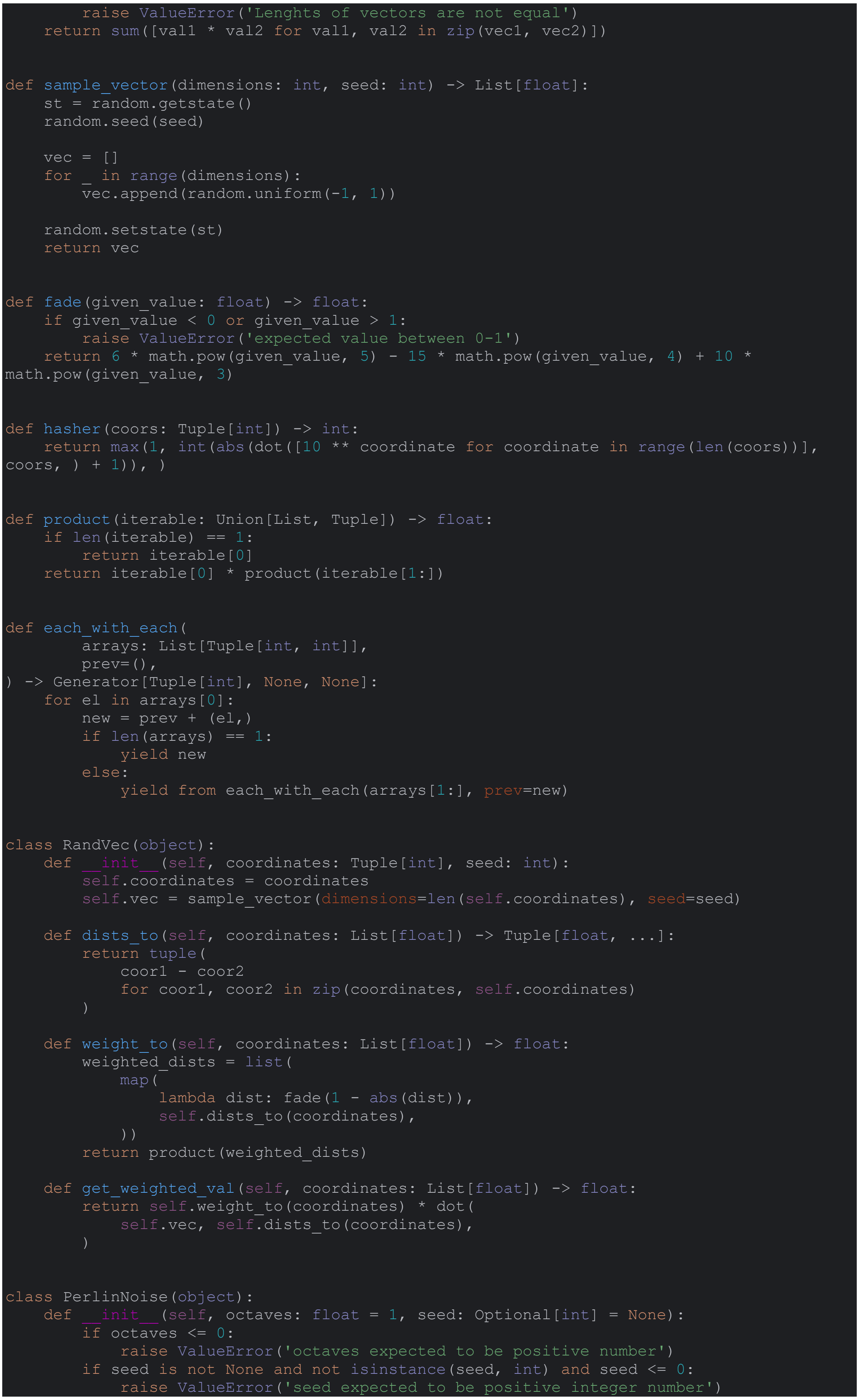
Variables for parameter values

Comment the unwanted seed variable for experiment

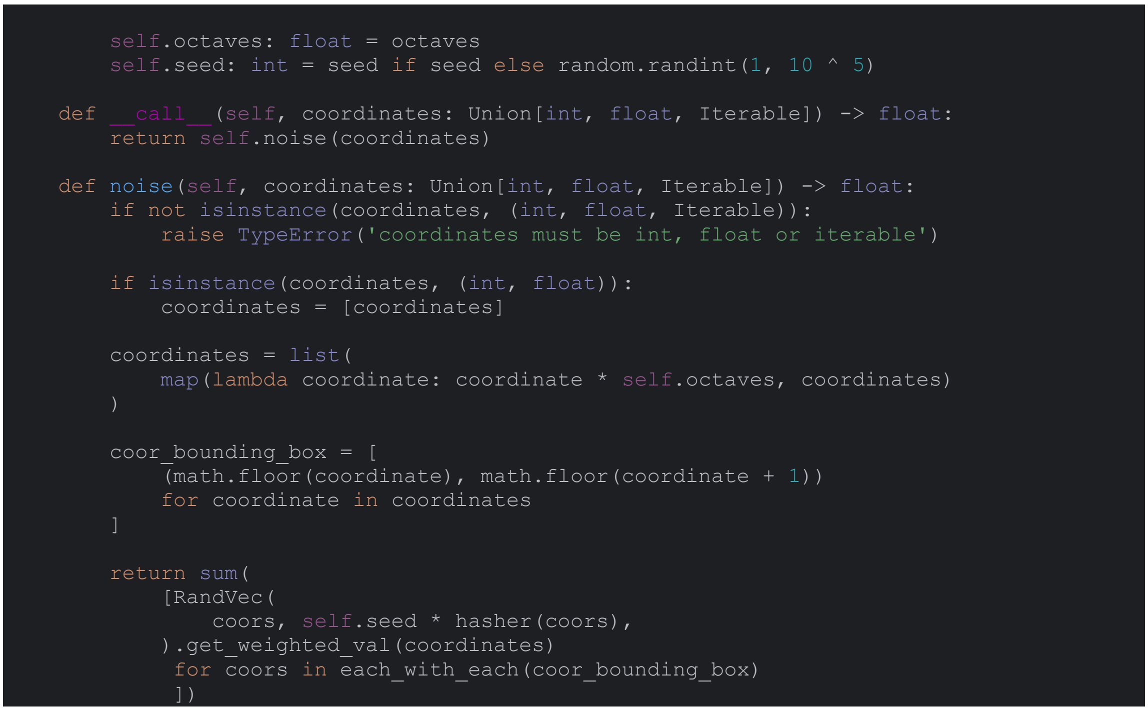
Perlin_Noise.py (Red Hen Dev *perlin_module.py*)



```
import math
import random
from _collections_abc import Iterable
from typing import Optional, Union, List, Tuple, Generator


def dot(
        vec1: Union[List, Tuple],
        vec2: Union[List, Tuple],
) -> Union[float, int]:
    if len(vec1) != len(vec2):
```



```
        raise ValueError('Lenghts of vectors are not equal')
    return sum([val1 * val2 for val1, val2 in zip(vec1, vec2)])


def sample_vector(dimensions: int, seed: int) -> List[float]:
    st = random.getstate()
    random.seed(seed)

    vec = []
    for _ in range(dimensions):
        vec.append(random.uniform(-1, 1))

    random.setstate(st)
    return vec


def fade(given_value: float) -> float:
    if given_value < 0 or given_value > 1:
        raise ValueError('expected value between 0-1')
    return 6 * math.pow(given_value, 5) - 15 * math.pow(given_value, 4) + 10 *
math.pow(given_value, 3)


def hasher(coors: Tuple[int]) -> int:
    return max(1, int(abs(dot([10 ** coordinate for coordinate in range(len(coors))],
coors, ) + 1)), )


def product(iterable: Union[List, Tuple]) -> float:
    if len(iterable) == 1:
        return iterable[0]
    return iterable[0] * product(iterable[1:])


def each_with_each(
        arrays: List[Tuple[int, int]],
        prev=(),
) -> Generator[Tuple[int], None, None]:
    for el in arrays[0]:
        new = prev + (el,)
        if len(arrays) == 1:
            yield new
        else:
            yield from each_with_each(arrays[1:], prev=new)


class RandVec(object):
    def __init__(self, coordinates: Tuple[int], seed: int):
        self.coordinates = coordinates
        self.vec = sample_vector(dimensions=len(self.coordinates), seed=seed)

    def dists_to(self, coordinates: List[float]) -> Tuple[float, ...]:
        return tuple(
            coor1 - coor2
            for coor1, coor2 in zip(coordinates, self.coordinates)
        )

    def weight_to(self, coordinates: List[float]) -> float:
        weighted_dists = list(
            map(
                lambda dist: fade(1 - abs(dist)),
                self.dists_to(coordinates),
            ))
        return product(weighted_dists)

    def get_weighted_val(self, coordinates: List[float]) -> float:
        return self.weight_to(coordinates) * dot(
            self.vec, self.dists_to(coordinates),
        )


class PerlinNoise(object):
    def __init__(self, octaves: float = 1, seed: Optional[int] = None):
        if octaves <= 0:
            raise ValueError('octaves expected to be positive number')
        if seed is not None and not isinstance(seed, int) and seed <= 0:
            raise ValueError('seed expected to be positive integer number')
```



```
        self.octaves: float = octaves
        self.seed: int = seed if seed else random.randint(1, 10 ^ 5)

    def __call__(self, coordinates: Union[int, float, Iterable]) -> float:
        return self.noise(coordinates)

    def noise(self, coordinates: Union[int, float, Iterable]) -> float:
        if not isinstance(coordinates, (int, float, Iterable)):
            raise TypeError('coordinates must be int, float or iterable')

        if isinstance(coordinates, (int, float)):
            coordinates = [coordinates]

        coordinates = list(
            map(lambda coordinate: coordinate * self.octaves, coordinates)
        )

        coor_bounding_box = [
            (math.floor(coordinate), math.floor(coordinate + 1))
            for coordinate in coordinates
        ]

        return sum(
            [RandVec(
                coors, self.seed * hasher(coors),
            ).get_weighted_val(coordinates)
             for coors in each_with_each(coor_bounding_box)
             ])
```