A Vector Calculus-Based Exploration of Hydraulic Erosion

Research Question:

To what extent can vector calculus be applied to model erosion and improve the realism of Minecraft's landscapes?

Subject: Mathematics

Word Count: approx. 3420

Table of Contents

A Vector Calculus-Based Exploration of Hydraulic Erosion

Introduction:

Minecraft is a wildly popular video game that first came out on May 17, 2009. One great feature of Minecraft was that every time a player started a new game, they would be able to have the game create an entirely new and unique world at the press of a button. The core flow of progression in the game would be retained regardless of how the world would turn out, but the experience would be sufficiently different each time to give Minecraft incredible replay value. However, the algorithm Minecraft uses to generate worlds has often been criticised for being quite strange. There are several odd artefacts that can be observed in the worlds generated by the algorithm and several natural features that are missing. Mountains in Minecraft worlds, for example, literally stick out like sore thumbs, having a vertical drop in all directions and essentially looking like very slightly deformed and/or elongated cubes that have simply been attached to the ground. Hills are also extremely steep and simply look like smaller but proportionally wider versions of the aforementioned mountains. Rivers are very shallow, have no banks, and unexpectedly get cut off before ever connecting to the ocean. There is no runoff from mountains and no sign that there ever was. While it is not necessarily a bad thing for video game worlds to have a foreign, even alien, aesthetic to them, such an aesthetic would still entail having signs of land-shaping processes such as weathering, erosion, and tectonic activity, as such

processes and/or their effects are clearly observable on the vast majority, if not all of the terrestrial planets that have ever been observed up close. The effects of said processes are simply not observable in Minecraft. More than giving its landscapes an unfamiliar look in a way that would inspire curiosity and awe, the handiwork of Minecraft's algorithm simply starts to feel half-baked and overly simplistic, which starkly contrasts the landscapes that have been observed here on Earth. This is unfortunate, considering that Minecraft worlds are clearly inspired by our own. After I really started to consider the lack of realism in the outputs of Minecraft's terrain generation algorithm, I realised the inexplicable desire to apply my mathematical knowledge to this problem.

At the time, I was playing around with some foundational concepts of calculus in order to better understand more advanced concepts and applications as well as trying to build connections between it and other areas of personal interest. Seeing that certain tools like partial differentiation and mathematical expressions used for modelling that I had seen in past math classes required something to be held fixed, I wondered about how to deal with everything being allowed to vary and, after some searching, stumbled upon vector calculus. Taking a bit of a deeper dive, I familiarised myself with the fundamentals of vector calculus, following along with a course to build these concepts up to greater heights and realised that vector calculus was a very fitting tool to apply to the Minecraft problem. This naturally led me to my research question: "To what extent can vector calculus be applied to model erosion and improve the realism of Minecraft's landscapes?"

Since calculus can describe the relationships between several quantities and how they all change and affect one another, it is quite effective in almost, if not all, physical applications of mathematics. From something as simple as Newton's second law in classical physics, to something as potentially confusing as the Schrödinger equations in quantum physics, to the Maxwell Equations that describe electromagnetism, to the Navier-Stokes equations that describe the behaviour of fluids, to modelling the weather, to more mundane-sounding things such as how and where garbage dumped in rivers disperses upon entering the ocean, calculus and vectors prove useful in an incredibly wide range of real-world problems.

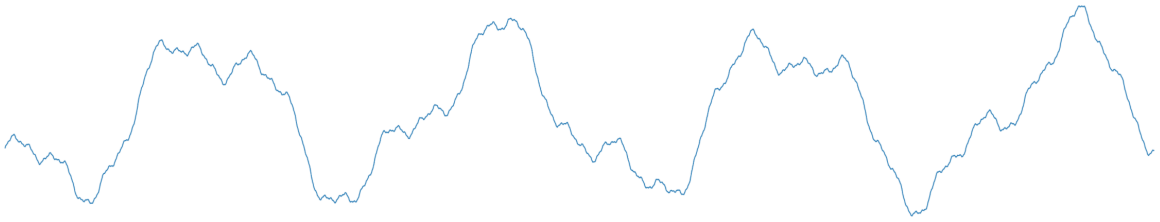
In this paper, my goal is to use vector calculus to build a model of the process of erosion from scratch and use it in combination with the algorithm on which Minecraft's is based to improve the terrain. Some elementary physics will be required to carry out this task, but I will only mention as much as is necessary to understand the thought process behind the formation and solution of the equation as this is a mathematics paper. My model will also be evaluated against that of Minecraft itself, which is essentially just purely procedural. This evaluation will be qualitative for the most part and focus on the presence of geographical features.

Two-Dimensional Case:

In order to understand how the water will flow in three spatial dimensions, it will help to first consider its behaviour in two dimensions. From this point forth, please refer to the Reference

Table that starts on page 29 to find the definitions of any variables and symbols that recur in this paper.

Consider an arbitrary function $f(x)$ that exists in the x/z plane describing the surface of a 2-dimensional cross-section of terrain that is continuous everywhere and infinitely differentiable and integrable. In addition, internal forces on the water due to its own pressure and viscosity, as well as external forces due to wind/air resistance are considered negligible in order to simplify the model:



Now, for a given volume of water at some point on the surface $(x, f(x))$ [note that ordered pairs will be of the form (x, z) since we are working on the x/z plane], we can create an expression for the net force (which simply becomes the sum of the gravitational and frictional forces when the conditions above are taken into account).

This gives us the following general expression:

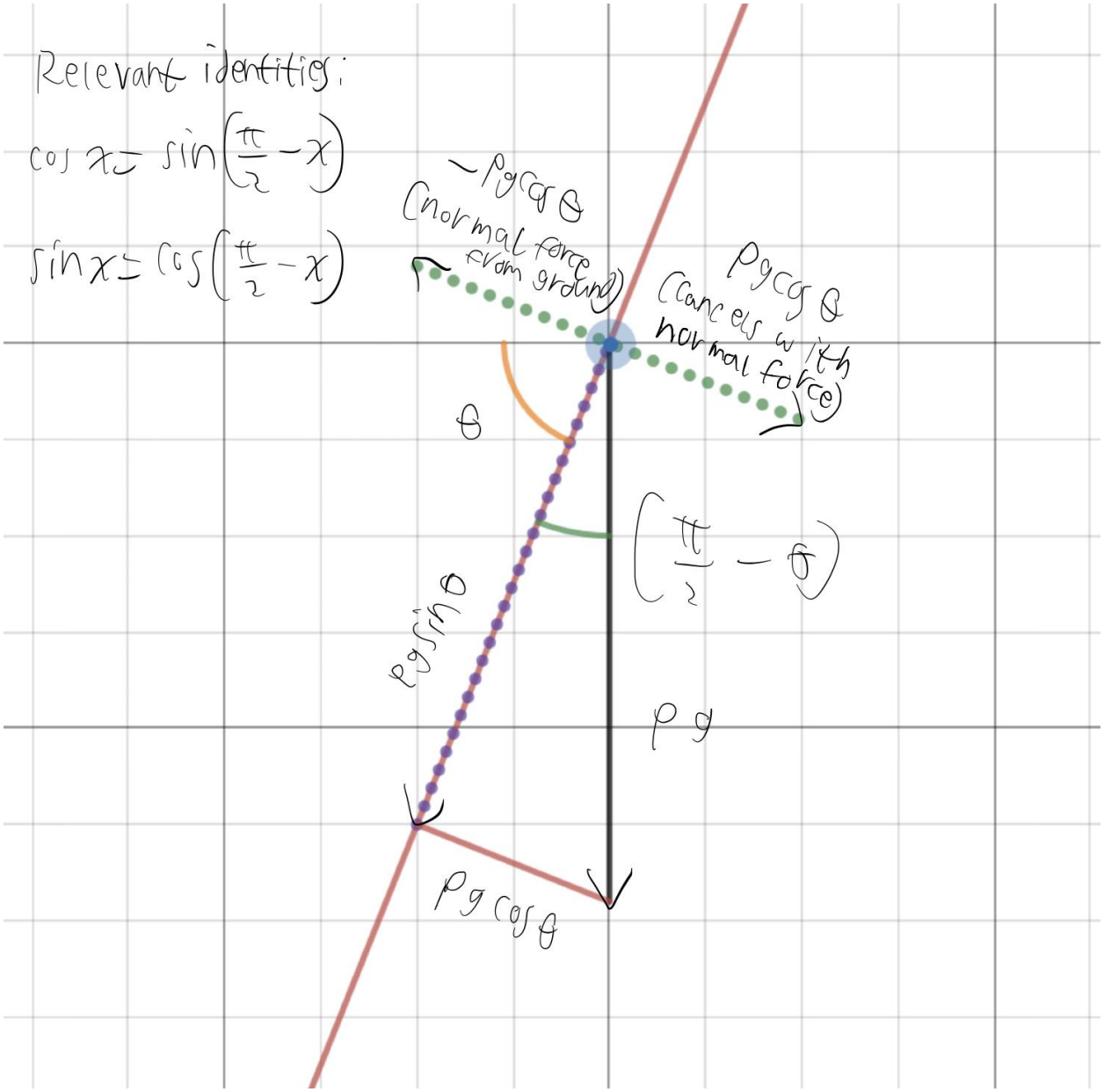
$$\vec{F_n} = \vec{f_k} + \vec{g} \tag{1}$$

Where $\vec{F_n}$ is the vector describing the net force, $\vec{f_k}$ is the force vector for friction, and $\vec{g}$ is the force vector for gravity. This is the same as the following vector sum:

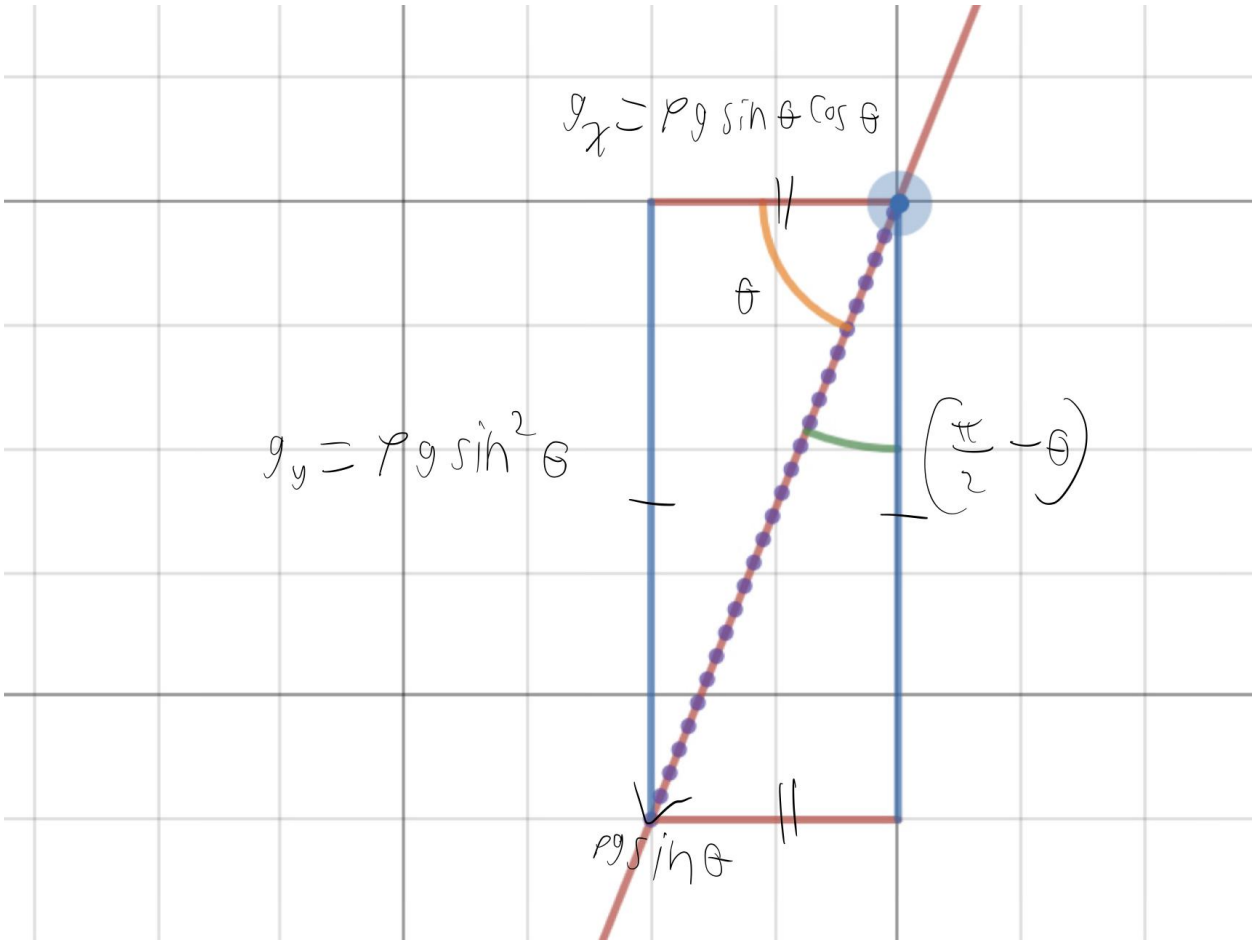
$$\rho\vec{a} = \rho g + k\mu\rho g\cos(\theta) \tag{2}$$

Forces are vectors, so it would make sense to turn this function into a vector-valued function to make the solution process easier. I will split the friction and gravity vectors into an x-component and a z-component so that I won't have to define a new coordinate system with new bases.

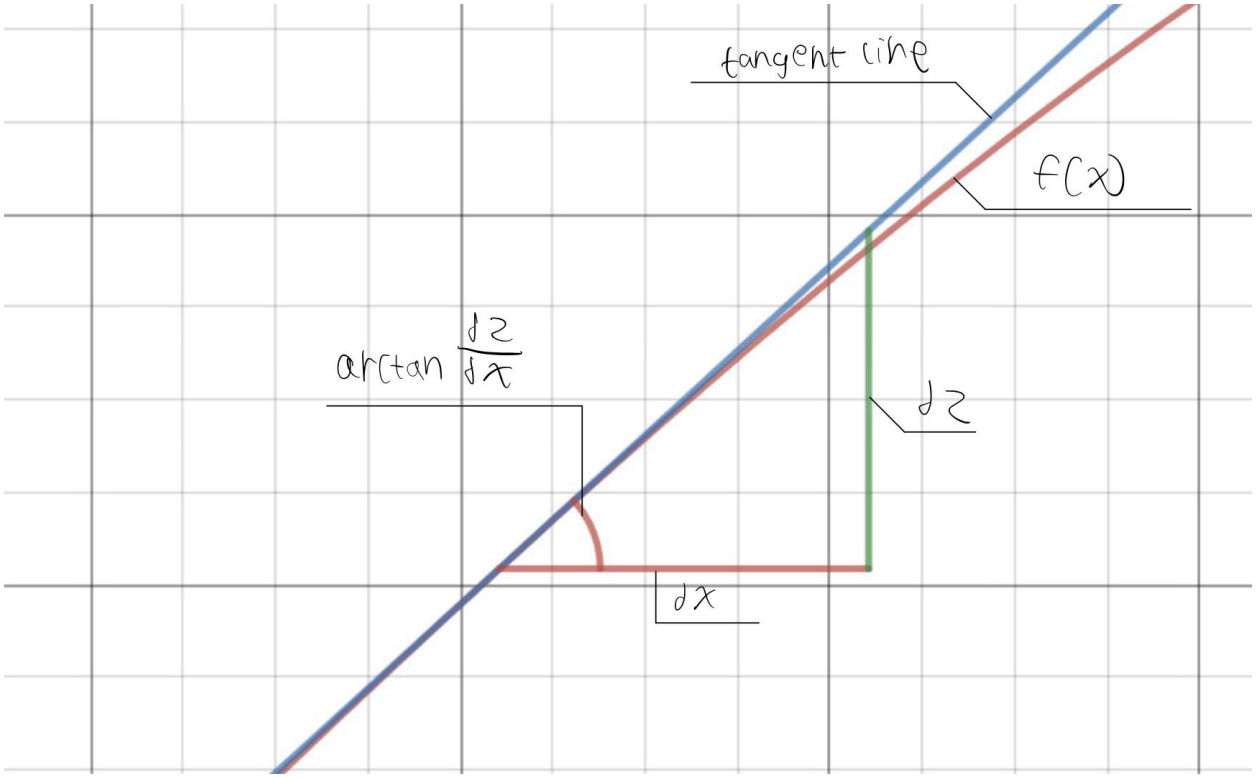
First, the force vector for gravity can be split into two components, one being normal to the tangent and the other parallel to it, yielding the following generic free body diagram for a unit volume of water:



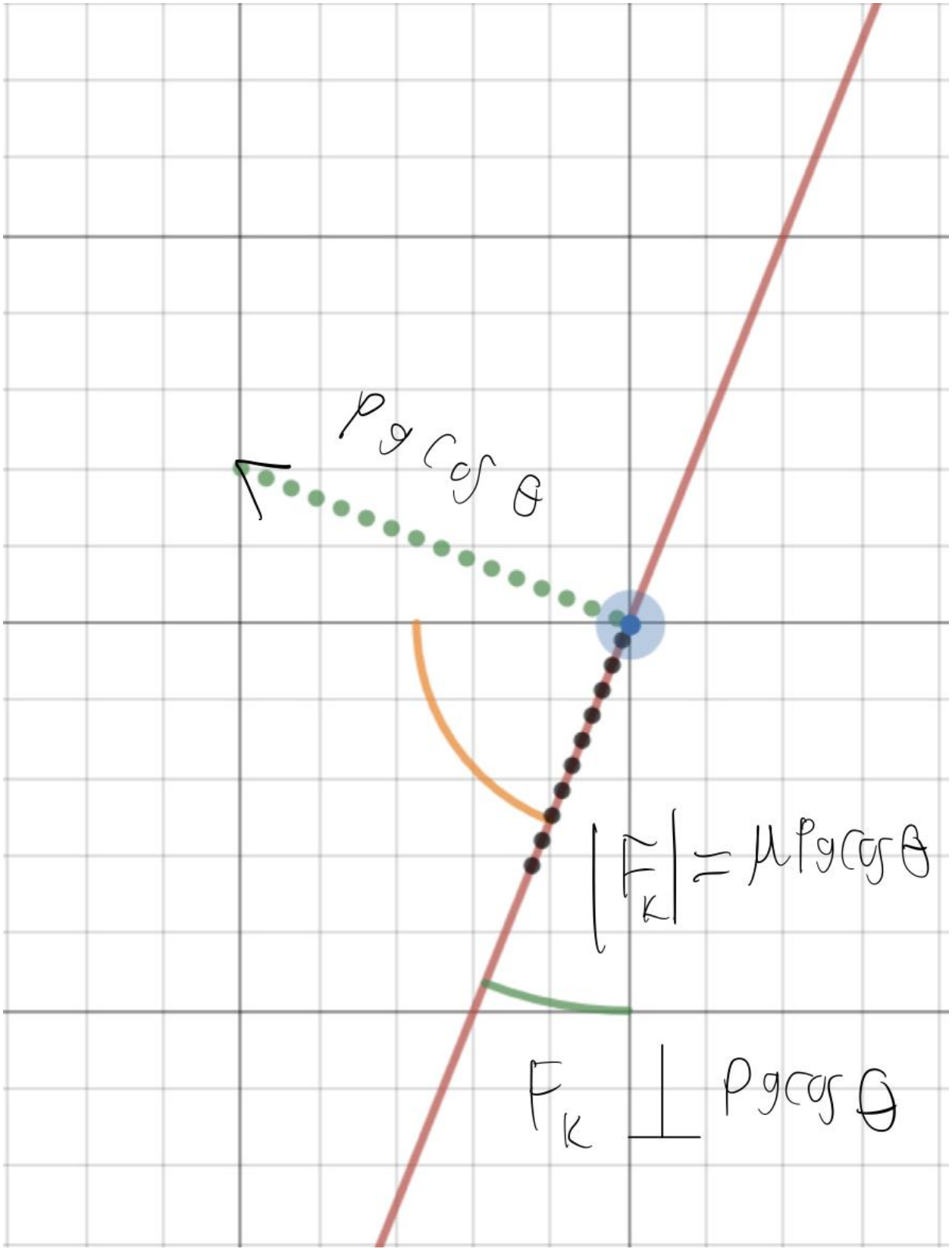
Note that the perpendicular component of gravitational force is cancelled out by the normal force the ground will exert on the water to counteract it. This is why objects on inclines will move along the incline and not away from or through them when no other external forces are exerted on them. This means we only need to worry about splitting the parallel component $\rho g \sin(\theta)$ into an x- and z- component. The next diagram shows this process:



The angle θ associated with a given point can be found by taking the arctangent of the derivative of $f(x)$:



Friction is determined using the normal component, even though it is fully counteracted by the contact/normal force from the ground:


$\rho g \cos\theta$
$|\vec{F}_k| = \mu \rho g \cos\theta$
$F_k \perp \rho g \cos\theta$

This can be split into two components using the same method as for gravity.

Using the components we've derived, we can build a vector-valued version of eq. (2):

$$\rho \begin{bmatrix} \vec{a_x} \\ \vec{a_z} \end{bmatrix} = \begin{bmatrix} \rho g \sin(\theta) \cos(\theta) + k\mu\rho g \cos^2(\theta) \\ \rho g \sin^2(\theta) + k\mu\rho g \cos(\theta) \sin(\theta) \end{bmatrix} \tag{3}$$

We know that the components of a, the acceleration, are simply the time derivatives of the corresponding components of the velocity, u.

This gives us the following equation:

$$\rho \begin{bmatrix} \frac{d\vec{u_x}}{dt} \\ \frac{d\vec{u_z}}{dt} \end{bmatrix} = \begin{bmatrix} \rho g \sin(\theta) \cos(\theta) + k\mu\rho g \cos^2(\theta) \\ \rho g \sin^2(\theta) + k\mu\rho g \cos(\theta) \sin(\theta) \end{bmatrix} \tag{4}$$

Solving out this equation will give us a way to find velocity vectors on the curve for all points $P = (x, f(x))$, which we can then use to help account for erosion.

It is important to remember that θ varies with respect to position, x, here, so introducing time into this equation through integration would add unnecessary complication. We can use a little algebraic manipulation to eliminate it and rewrite the derivative as one with respect to x:

$$\rho \begin{bmatrix} \frac{dx}{dx} & \frac{d\vec{u_x}}{dt} \\ \frac{dz}{dz} & \frac{d\vec{u_z}}{dt} \end{bmatrix} = \begin{bmatrix} \rho g \sin(\theta)\cos(\theta) + k\mu\rho g \cos^2(\theta) \\ \rho g \sin^2(\theta) + k\mu\rho g \cos(\theta)\sin(\theta) \end{bmatrix} \tag{5}$$

The differentials can be treated as regular numbers here so we can use the commutative property of multiplication to "swap" the dx in the denominator with dt:

$$\rho \begin{bmatrix} \frac{dx}{dt} & \frac{d\vec{u_x}}{dx} \\ \frac{dz}{dt} & \frac{d\vec{u_z}}{dz} \end{bmatrix} = \begin{bmatrix} \rho g \sin(\theta)\cos(\theta) + k\mu\rho g \cos^2(\theta) \\ \rho g \sin^2(\theta) + k\mu\rho g \cos(\theta)\sin(\theta) \end{bmatrix} \tag{6}$$

$\frac{dx}{dt}$ and $\frac{dz}{dt}$ are simply u_x and u_z respectively, so our new equation is this:

$$\rho \begin{bmatrix} \vec{u_x} & \frac{d\vec{u_x}}{dx} \\ \vec{u_z} & \frac{d\vec{u_z}}{dz} \end{bmatrix} = \begin{bmatrix} \rho g \sin(\theta)\cos(\theta) + k\mu\rho g \cos^2(\theta) \\ \rho g \sin^2(\theta) + k\mu\rho g \cos(\theta)\sin(\theta) \end{bmatrix} \tag{7}$$

To solve for the velocity vector field for a unit volume of water, we simply go component-by-component and separate the derivative to integrate. In order to further clean up the expression, I will also divide through by ρ since it is a common factor of every term:

$$\begin{bmatrix} \vec{u_x} & \frac{d\vec{u_x}}{dx} \\ \vec{u_z} & \frac{d\vec{u_z}}{dz} \end{bmatrix} = \begin{bmatrix} g \sin(\theta)\cos(\theta) + k\mu g \cos^2(\theta) \\ g \sin^2(\theta) + k\mu g \cos(\theta)\sin(\theta) \end{bmatrix}$$

$$\begin{bmatrix} \vec{u_x} d\vec{u_x} \\ \vec{u_z} d\vec{u_z} \end{bmatrix} = \begin{bmatrix} (g\sin(\theta)\cos(\theta) + k\mu g\cos^2(\theta))dx \\ (g\sin^2(\theta) + k\mu g\cos(\theta)\sin(\theta))dz \end{bmatrix}$$

$$\begin{bmatrix} \int \vec{u_x} d\vec{u_x} \\ \int \vec{u_z} d\vec{u_z} \end{bmatrix} = \begin{bmatrix} \int (g\sin(\theta)\cos(\theta) + k\mu g\cos^2(\theta))dx \\ \int (g\sin^2(\theta) + k\mu g\cos(\theta)\sin(\theta))dz \end{bmatrix} \tag{8}$$

As an aside, recall the definition of θ:

$$\begin{bmatrix} \int \vec{u_x} d\vec{u_x} \\ \int \vec{u_z} d\vec{u_z} \end{bmatrix} = \begin{bmatrix} \int (g\sin(\arctan(\frac{df}{dx}))\cos(\arctan(\frac{df}{dx})) + k\mu g\cos^2(\arctan(\frac{df}{dx}))dx \\ \int (g\sin^2(\arctan(\frac{df}{dx})) + k\mu g\cos(\arctan(\frac{df}{dx}))\sin(\arctan(\frac{df}{dx}))dz \end{bmatrix}$$

Now, isolating $\vec{u_x}$ and $\vec{u_z}$ and integrating the LHS from eq. (8) component-by-component and including the constants of integration (as well as absorbing anything unnecessary into them) gives us the following:

$$\begin{bmatrix} \frac{1}{2}\vec{u_x}^2 \\ \frac{1}{2}\vec{u_z}^2 \end{bmatrix} = \begin{bmatrix} \int (g\sin(\theta)\cos(\theta) + k\mu g\cos^2(\theta))dx \\ \int (g\sin^2(\theta) + k\mu g\cos(\theta)\sin(\theta))dz \end{bmatrix} + \begin{bmatrix} C_x \\ C_z \end{bmatrix}$$

$$\begin{bmatrix} \vec{u_x} \\ \vec{u_z} \end{bmatrix} = \sqrt{2\begin{bmatrix} \int (g\sin(\theta)\cos(\theta) + k\mu g\cos^2(\theta))dx \\ \int (g\sin^2(\theta) + k\mu g\cos(\theta)\sin(\theta))dz \end{bmatrix} + \begin{bmatrix} C_x \\ C_z \end{bmatrix}} \tag{9}$$

Unfortunately, the presence of the nested trigonometric functions (especially the arctangent) leads to highly complex non-elementary antiderivatives which I do not have the knowledge necessary to compute precisely. The antiderivative will also look different depending on the function describing the terrain. Letting $f(x)$ equal something as simple as $-\cos(x)$, for example, and then computing just the antiderivative of $\arctan(\sin(x))$ yields a result that is too long to fit on a single line on this page and involves multiple instances of a non-elementary function known as the dilogarithm, which in this case would also have complex numbers as inputs. For the sake of simplicity, efficiency, and the focus of this paper, I will rely on a computer to determine them precisely.

One thing to note here is that the vector field resulting from this equation will look different depending on the initial conditions. Since every unit volume of water that is placed on the map when representing rainfall will be given a random initial velocity and random initial position, we will have to at least partially compute the vector field to know where the water will travel.

Before continuing this in the 3-dimensional case, it would be prudent to lay out additional conditions to simplify the next steps:

1) The velocity vector field is only defined for points on the surface described by the function f. We will also assume that the water is always in contact with the ground, so it does not need to be defined for other points.
2) The water is arbitrarily shallow. We are only using this condition to simplify things for now.

Three-Dimensional Case:

In three dimensions, the ground will be the x/y plane instead of the x-axis. The arbitrary function f describing the terrain will now take both x and y as inputs. The set of points describing the terrain in three dimensions will thus be of the form $P = (x, y, f(x, y))$. At any point on the surface representing the terrain, the only external forces acting on a droplet will once again be gravity and friction while all internal forces are considered negligible.

One more thing to note will be that the assumptions that the water is always in contact with the ground and that it is arbitrarily shallow greatly simplify the solution process in 3 dimensions as this will remove the z-component from the picture. As far as the simulation will be concerned, the z-component of the water's position will always be equal to the value of $f(x, y)$ at that position.

At any point on the surface, we can find a tangent line for the y/z slope and the x/z slope by computing the partial derivatives of f with respect to y and x, respectively [they will henceforth be referred to as f_x and f_y]. The arctangents of these partial derivatives will give us the angles θ_x and θ_y respectively of the corresponding tangent lines from the x/y plane.

Once again, the normal/contact force exerted on the droplet by the terrain is cancelled out by the component of the gravitational force that is perpendicular to the terrain at that point. Thus, the

only force vectors that remain are $\rho g \sin(\theta_x)$ and $\rho g \sin(\theta_y)$ and we must split each into its x- and y-components and then build up to an equation resembling eq. (9)

Recalling the process we used in the two-dimensional case in eq. (3), we can construct the following expressions:

$$\rho g_x = \rho g \sin(\arctan(f_x)) \cos(\arctan(f_x)) \tag{10.1}$$

$$\rho g_y = \rho g \sin(\arctan(f_y)) \cos(\arctan(f_y)) \tag{10.2}$$

The expressions for friction will be as follows:

$$\rho \overrightarrow{f_{k\,x}} = k\mu g \cos^2(\arctan(f_x)) \tag{11.1}$$

$$\rho \overrightarrow{f_{k\,y}} = k\mu g \cos^2(\arctan(f_y)) \tag{11.2}$$

Putting the corresponding components of eq. (10) and (11) together, we get the following:

$$\rho \frac{d\vec{u}}{dt} = \begin{bmatrix} \rho g_x + \rho \vec{f_{k}}_x \\ \rho g_y + \rho \vec{f_{k}}_y \end{bmatrix}$$

Dividing through by ρ:

$$\frac{d\vec{u}}{dt} = \begin{bmatrix} g_x + \vec{f_{k}}_x \\ g_y + \vec{f_{k}}_y \end{bmatrix} \tag{12}$$

Rewriting the derivative:

$$\begin{bmatrix} \frac{d\vec{u_x}}{dt} \\ \frac{d\vec{u_y}}{dt} \end{bmatrix} = \begin{bmatrix} g_x + \vec{f_{k}}_x \\ g_y + \vec{f_{k}}_y \end{bmatrix}$$

$$\begin{bmatrix} \frac{dx}{dx} & \frac{d\vec{u_x}}{dt} \\ \frac{dy}{dy} & \frac{d\vec{u_y}}{dt} \end{bmatrix} = \begin{bmatrix} g_x + \vec{f_{k\,x}} \\ g_y + \vec{f_{k\,y}} \end{bmatrix}$$

$$\begin{bmatrix} \frac{dx}{dt} & \frac{d\vec{u_x}}{dx} \\ \frac{dy}{dt} & \frac{d\vec{u_y}}{dy} \end{bmatrix} = \begin{bmatrix} g_x + \vec{f_{k\,x}} \\ g_y + \vec{f_{k\,y}} \end{bmatrix}$$

$$\begin{bmatrix} \vec{u_x} & \frac{d\vec{u_x}}{dx} \\ \vec{u_y} & \frac{d\vec{u_y}}{dy} \end{bmatrix} = \begin{bmatrix} g_x + \vec{f_{k\,x}} \\ g_y + \vec{f_{k\,y}} \end{bmatrix}$$

$$\begin{bmatrix} \vec{u_x} d\vec{u_x} \\ \vec{u_y} d\vec{u_y} \end{bmatrix} = \begin{bmatrix} (g_x + \vec{f_{k\,x}})dx \\ (g_y + \vec{f_{k\,y}})dy \end{bmatrix}$$

$$\begin{bmatrix} \int \vec{u_x} d\vec{u_x} \\ \int \vec{u_y} d\vec{u_y} \end{bmatrix} = \begin{bmatrix} \int (g_x + \vec{f_{k\,x}})dx \\ \int (g_y + \vec{f_{k\,y}})dy \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{2}\vec{u_x}^2 \\ \frac{1}{2}\vec{u_y}^2 \end{bmatrix} = \begin{bmatrix} \int (g_x + \vec{f_{k}}_x)dx \\ \int (g_y + \vec{f_{k}}_y)dy \end{bmatrix} + \begin{bmatrix} C_x \\ C_y \end{bmatrix}$$

$$\begin{bmatrix} \vec{u_x} \\ \vec{u_y} \end{bmatrix} = \sqrt{2 \begin{bmatrix} \int (g_x + \vec{f_{k}}_x)dx \\ \int (g_y + \vec{f_{k}}_y)dy \end{bmatrix} + \begin{bmatrix} C_x \\ C_y \end{bmatrix}} \tag{13}$$

Now that the necessary expressions have been derived, they will be implemented in the next section.

Erosion:

Clearly, the mass and velocity of the water will have some effect on both the water's capacity and the amount of terrain that is eroded at a certain point. The amount of earth already being carried by the water will determine how much more the water can erode without having to deposit some of it. Certain characteristics of the terrain the water is going over will also have an effect. However, I have insufficient knowledge of this to be able to build this equation by myself.

This segment is where the secondary research plays its part. I will be incorporating some principles from two other implementations of hydraulic erosion, namely Hans Theobald Beyer's "Implementation of a method for hydraulic erosion" and Sebastian Lague's video "Coding Adventure: Hydraulic Erosion," which is actually loosely based off of the former.

"Implementation of a method for hydraulic erosion" mentions two rules used in the simulation which are of particular mathematical importance:

1) While not fully saturated, droplets will pick up a percentage of their remaining sediment capacity in each step. While oversaturated, droplets will deposit a percentage of the surplus sediment in each step.
2) The water's sediment capacity c is the product of the water's mass and velocity, the slope of the ground on which it travels, and a physical constant essentially representing a fluid's ability to erode.

Statements 3 and 5 are of particular mathematical significance. The rest are more significant to the simulation, which is not the focus of this paper.

According to statement 5:

$$c = n(slope)(mass)(velocity)$$

where h is the physical constant. $(velocity)$ is simply going to be the $\vec{u}$ that we solved for. I will shorten $(mass)$ to m and replace $(slope)$ with the droplet's change in height during a step d (Beyer). This makes the expression more compact:

$$c = h * d * m * u \tag{14}$$

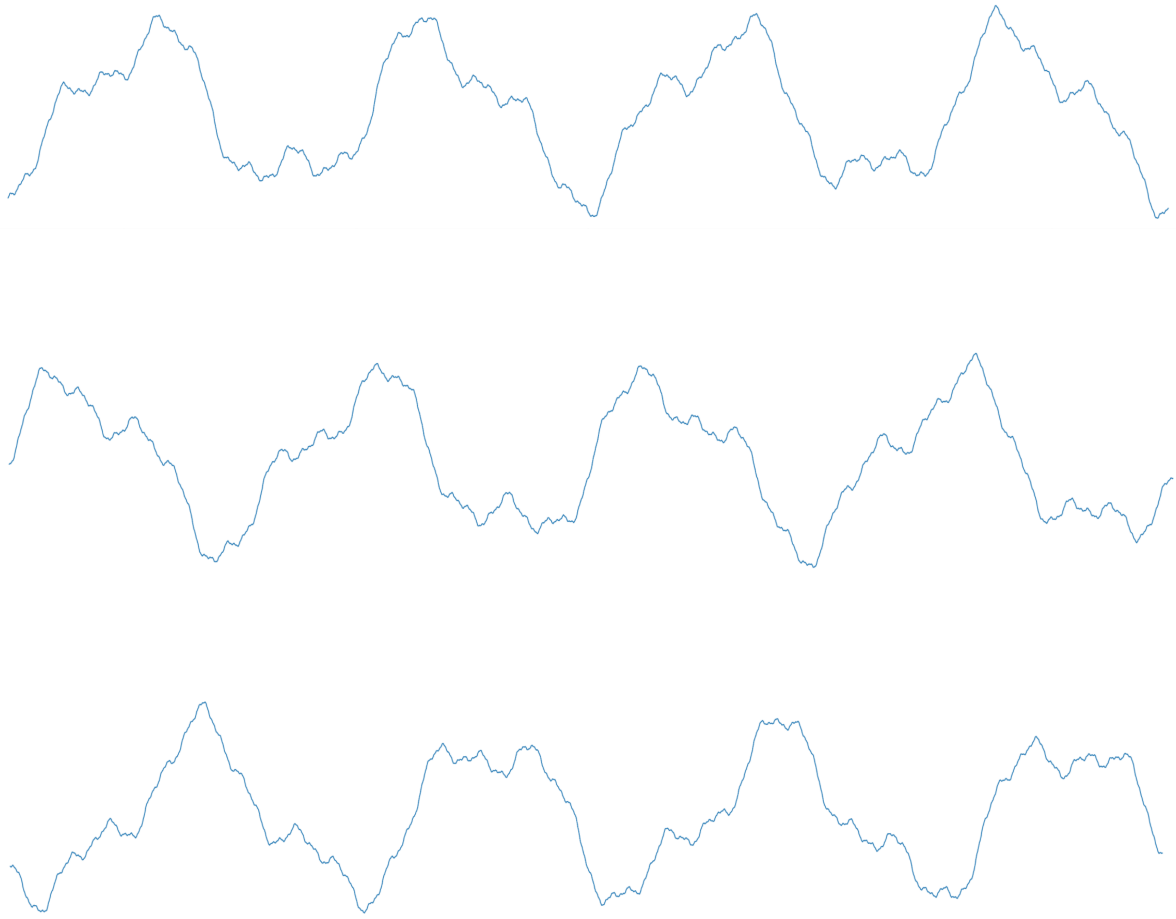
Now we will write an expression representing the third statement. I will define a new variable p as the amount of sediment currently being carried and a variable n as a number in the interval $[0, 1)$. With this, we can write a piecewise relation for p:

$$\mathrm{p}_{final} = \begin{cases} p_{initial} + nc, & p_{initial} \leq c \\ p_{initial} - n(p_{initial} - c), & p_{initial} > c \end{cases} \tag{15}$$

Initial Heightmap:

With all of the relevant expressions in place, we can finally start to implement them. With the help of "FractalNoiseFunctions," I built a simple 2-dimensional Perlin Noisemap generator. Following the example of the source, I created an iterator that adds together up to 10 sinusoidal functions. The frequency of each successive sinusoid grows exponentially while the wavelength and amplitude decay exponentially. The nature of this growth can be controlled by the variables persistence (which corresponds to the decay in amplitude) and lacunarity (which corresponds to

the decay in wavelength and the growth in frequency) (Lague). Shown below are a few samples of terrain that resulted from this:



To adapt my mathematical work to this, there are a couple of steps that must be taken and rules that must be implemented to streamline implementation (some of which are arbitrary and mostly for the sake of simplicity):

1) All droplets will have a mass of $0.005kg$ and thus a volume of $5mL$ (due to the density of water). This will be held constant.

2) The initial speed of the droplets will be randomly selected from a small interval between 0 and $1ms^{-1}$. They will travel along the terrain in either the positive or negative x -direction.

This will be expanded on in the next section.

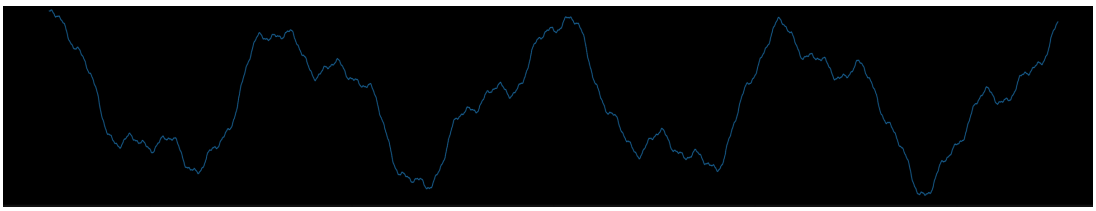
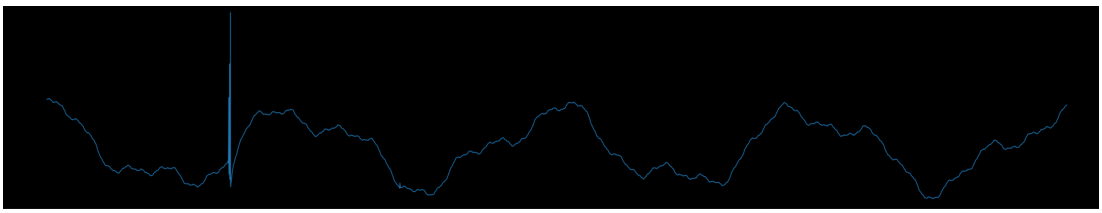
Discretisation:

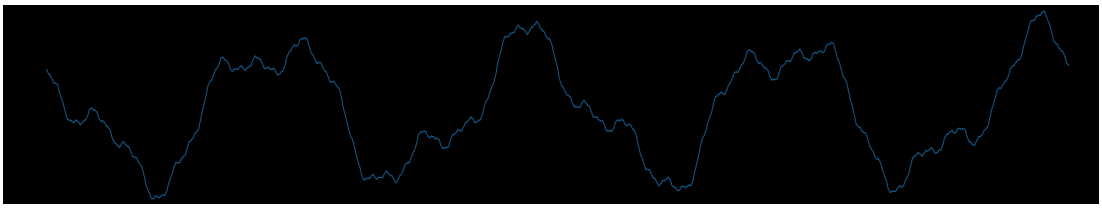
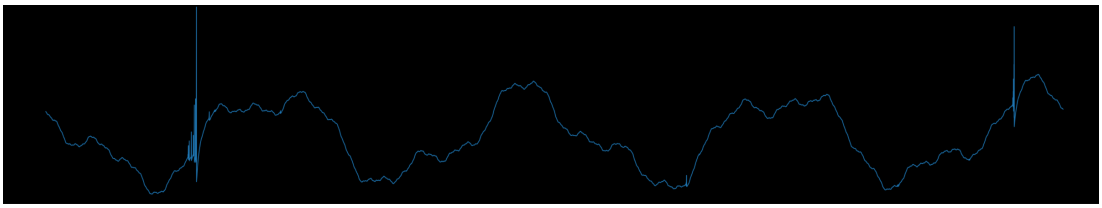
To properly run this simulation, discretisation is required. In order to discretise this problem, I have used Python to sample 10,000 evenly-spaced points on the interval [-5, 5] that all satisfy a heightmap generated using the aforementioned simple algorithm and draw a curve through them. The resulting curve is effectively indistinguishable from what a continuous plot of the function would look like but is in reality discrete. This gives us "cells" on the heightmap that we can alter using the expressions we've derived above. For the sake of brevity, any distances mentioned from this point on will be talked about in terms of cells.

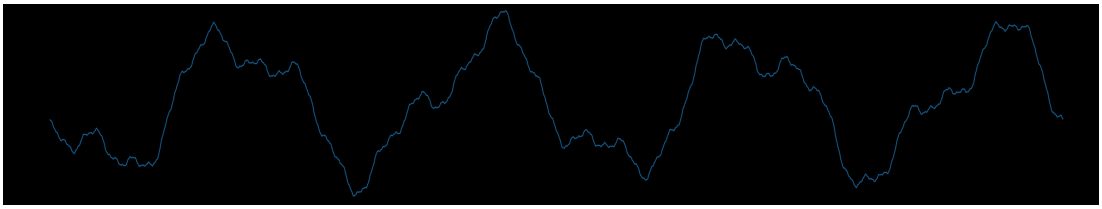
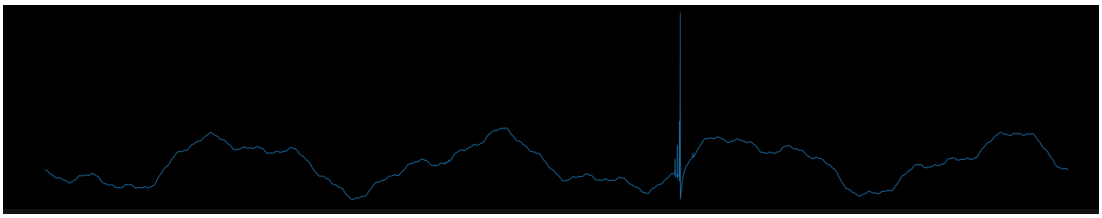
The behaviour of the droplets in my implementation borrows heavily from Section 5 of Hans Theobald Beyer's "Implementation of a method for hydraulic erosion." I've effectively just slightly altered a few of the rules so as to better represent the equations I've derived. For example, all droplets are initialised at a random cell with a random speed of up to 5 cells per iteration either to the left or right, and will not be able to travel faster than 30 cells per iteration (this is made to be within the range of the speeds of most rivers around the world). Adhering to

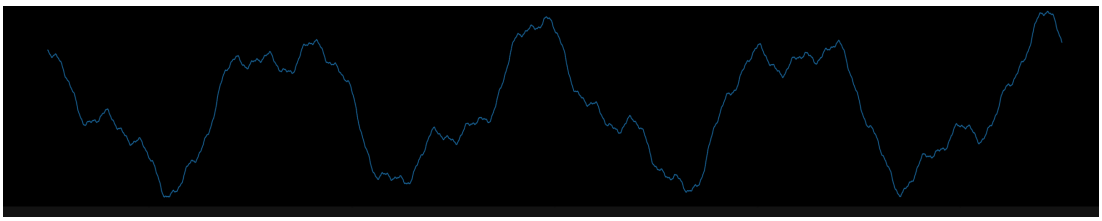
the simplification that the water is always in contact with the terrain, the droplets' y-positions are always equal to the value of the heightmap at their x-position. In each iteration, droplets will move a certain number of cells based on both their speed and the "gradients" between the adjacent cells that droplets travel through. Every time a droplet crosses a cell, its speed (and possibly direction) may change based on the gradient it moved through. In order to ensure droplets only cross an integer number of cells each iteration, the droplet's speed will not change if the work done on the droplet across the gradient is insufficient. The work done across a gradient is calculated only in the x-direction due to the simplification regarding droplets' y-positions. If a droplet gets "stuck" anywhere on the heightmap, it will be removed. If not, it will automatically "evaporate" after a fixed number of iterations.

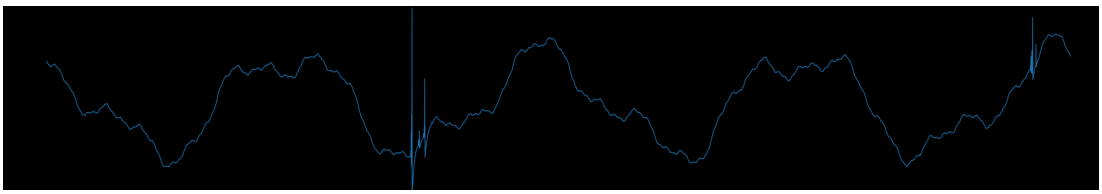
The first few trials resulted in negligible erosion, but after fixing a lot of minor errors that were getting in the way of the process, I managed to mostly fix the algorithm and came up with some of these (admittedly slightly flawed) samples:

Before	
After	

Before	
After	

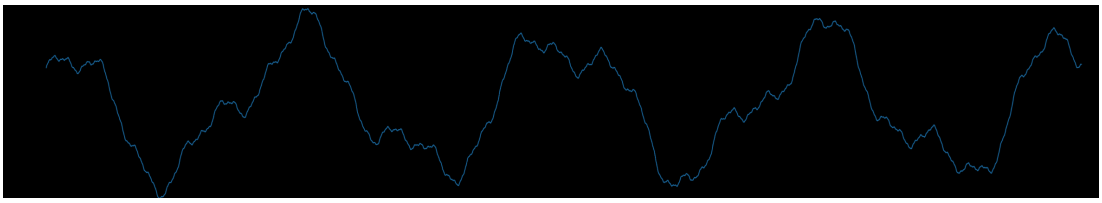
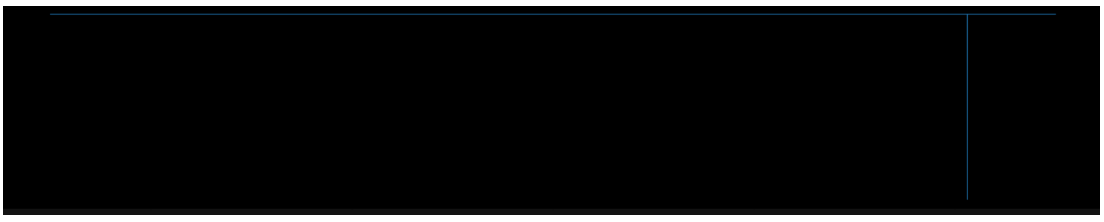
Before	
After	

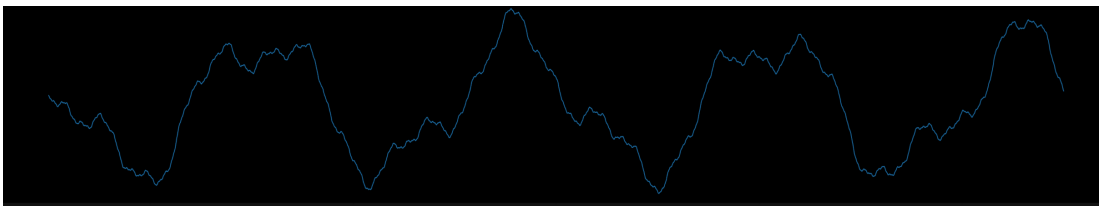
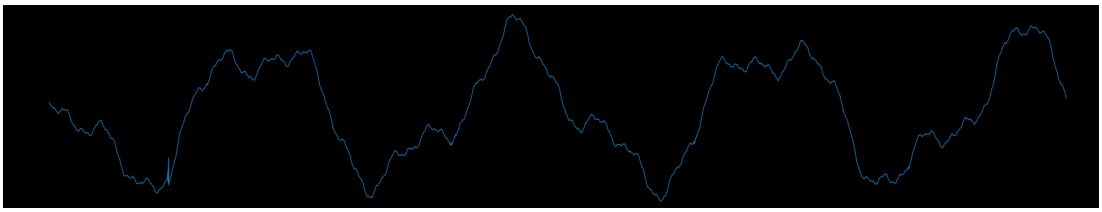
Before	

After	

Evaluation:

The erosion process can result in noticeable changes to the terrain but for the moment requires much revision. The most notable issues are the spikes that occasionally appear on the terrain due to droplets getting “stuck” in a certain area and falling into a cycle of erosion and deposition without any actual movement and the relative lack of consistency in the simulations. Sometimes, no noticeable erosion occurs because most of the droplets evaporate after getting “stuck”, and other times, too much occurs. Two examples of this are shown below:

Before	
After	

Before	
After	

It also only works in 2 dimensions so far. However, it does have potential. The pure procedural approach itself already generates decent terrain, the artifacts of which are not very noticeable on a small scale, and rectifying this with erosion is certainly promising. The next steps involve extending this to 3 dimensions and adding more variation to the droplet behaviour by making sure the ground is not just a giant mass of exactly one material. The implementation could also use some streamlining to be more effective and run faster. Finally, it is clearly observable that the algorithm simply reduces the relief of the terrain but does not do a great job of actually smoothing it out. In addition, some degree of periodicity can be seen in each sample before erosion was simulated in spite of the smaller scale of the samples since all the details are added over the parent function $\cos(x)$. Even with all their flaws, I do think the results warrant an affirmative response to the research question, however.

Conclusion:

Considering the results, vector calculus does have use in modelling erosion to augment the realism of procedurally generated terrain despite the flaws that turned out in the approach taken here. Erosion naturally leads to fractal-like effects that resemble those in real life which also help mitigate artifacts of procedural methods. Despite the erosion really only affecting the relief of the terrain in the samples taken from my work, it did eliminate any observable periodicity. As for extending to three dimensions, it seems that a good way to extend the generation method itself would be to add together terms that are composed of a sinusoid in x multiplied by a sinusoid in y. It would also require using the directional derivative of the resulting function to alter the velocity of the water in each step. In spite of the exploration not turning out to be completely successful, it did help to reveal some information about how to better adapt continuous, physics- and vector-based models to the discrete nature of Minecraft to obtain good approximations, as well as laying a foundation to build upon for the next steps. The idea of realistic Minecraft terrain is definitely not just wishful thinking.

Reference Table (for recurring symbols and variables):

Symbol/Variable	Definition/Explanation
x, y, and z	x, y, and z (where applicable) are the spatial dimensions used in this investigation. In 2 dimensions, z represents height and the x-axis represents the ground. In 3 dimensions, z still represents height but the ground is now the x/y plane for all $z = 0$. In the equations, these variables represent the positions of a unit volume of water. Note that any subscripts involving one of these symbols indicate the corresponding component of the variable in question.
$f(x)$	The arbitrary function that represents a 2-dimensional sample of terrain. It is continuous everywhere and infinitely differentiable and integrable because it is defined as a sum/difference of sinusoidal

	functions.
$f(x,\ y)$	The arbitrary function representing a 3-dimensional sample of terrain. It is continuous everywhere and infinitely differentiable and integrable because it is defined as a sum/difference of products/quotients of sinusoidal functions that depend on x and/or y.
g	The acceleration due to gravity. It has a magnitude of $9.81\ m\ s^{-2}$ and always points straight down towards the x-axis (in 2 dimensions) and the x/y plane (in 3 dimensions)
$\vec{g}$	The vector representing gravitational force. It is defined as ρg.
k	A constant introduced in the definition of the frictional force. k has a value of either 1 or -1 depending on the direction in which a given volume of water is travelling since friction always opposes motion. Dimensionless.

μ	The coefficient of kinetic friction between the ground and the water. Dimensionless.
ρ	The density of water, $997\ kg\ m^{-3}$.
$\vec{a}$	The acceleration vector for a given unit volume of water at a given point $(x,\ f(x))$ or $(x,\ y,\ f(x,\ y))$. This is a vector-valued function and the time-derivative of $\vec{u}$.
θ	In 2 dimensions, the angle of the tangent line to $f(x)$ at x from the horizontal. This can be found by taking the arctangent of the derivative at the given point.
θ_x	In 3 dimensions, the angle of the tangent line described by $\frac{\partial f}{\partial x}$ to the x/y plane (which represents the "horizontal").
θ_y	In 3 dimensions, the angle of the tangent line described by $\frac{\partial f}{\partial y}$ to the x/y plane (which represents the "horizontal").
t	Time.

$\vec{u}$	A vector-valued function representing the velocity of a volume of water at any point $(x,\ f(x))$ or $(x,\ y,\ f(x,\ y))$.

Works Cited

Beyer, Hans Theobald, "Implementation of a method for hydraulic erosion." TECHNISCHE UNIVERSITÄT MÜNCHEN DEPARTMENT OF INFORMATICS, 15 November 2015, https://www.firespark.de/resources/downloads/implementation%20of%20a%20methode%20for%20hydraulic%20erosion.pdf. Accessed 30 April 2020.

“Coding Adventure: Hydraulic Erosion." YouTube, uploaded by Sebastian Lague, 28 Feb 2019, https://www.youtube.com/watch?v=eaXk97ujbPQ.

“Procedural Landmass Generation (E01: Introduction).” YouTube, uploaded by Sebastian Lague, 31 Jan 2016, https://www.youtube.com/watch?v=wbpMiKiSKm8&list=PLFt_AvWsXl0eBW2EiBtl_sxmDtSgZBxB3&index=1.

“FractalNoiseFunctions.” Desmos, https://www.desmos.com/calculator/u5znu9c6ny. Accessed 1 Nov 2020.

Appendix: Simulation Code (with Sample Output)

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib as mpl
import random as rand
plt.rcParams['axes.facecolor'] = 'black'
```

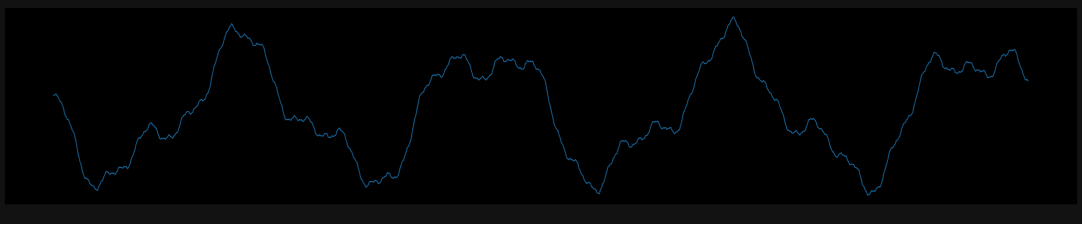
In [2]:
```
def perlin(x):
    return np.cos(x)
```

In [3]:
```
persistence = np.e
lacunarity = 2.5
```

In [4]:
```
x = np.linspace(-5,5,10000)
```

In [5]:
```
y = np.linspace(0,0,10000)
for i in range(1,11):
    y += 10*perlin((lacunarity**i)*x + rand.random()*rand.random()*10.24)/(persistence**i)+0.6
plt.figure(figsize=(256,48))
plt.plot(x,y,linewidth=10)
```

Out[5]:
```
[<matplotlib.lines.Line2D at 0x1d37a9cbac8>]
```



In [6]:
```
 droplet:
lef __init__(self):
    xpos = rand.randint(50,9999-50)
    self.pos = [xpos, y[xpos]]
    self.gradients = [[1, y[xpos+1]-y[xpos]],[-1,y[xpos-1]-y[xpos]]]
    self.angles = [np.arctan(self.gradients[0][1]/self.gradients[0][0]),np.arctan(self.gradients[1][1]/self.gradients[1][0])]
    self.load = 0
    self.speed = rand.randint(-30,30)
    self.mass = 0.005
    self.volume = 0.005
    self.mu_k = 0.160
    self.max_speed = 30
    self.e_const = 1
    self.capacity = 0
    self.p_erosion = 0.5
    self.p_deposition = 0.5
```

In [7]:
```
iterations = 8
droplets = 100
```

In [ ]:

```
from IPython.core.debugger import set_trace
output = y
friction_dir=0
e_custom = 0.5
h_diff = 0
it_num = 0

for h in range(0,droplets+1):
    d=droplet()
    vel=d.speed
    for i in range(0,iterations+1):
        previous_positions = [None]*10
        empty = 0
        steps = abs(vel)
        total_steps = steps
        if vel == 0:
            vel = min(d.gradients[0][1],d.gradients[1][1])/abs(min(d.gradients[0][1],d.gradients[1][1]))
            steps = abs(vel)
        while steps > 0:
            if (d.pos[0] in range(50,9999-50)):
                for k in range(1,10):
                    previous_positions[-k] = previous_positions[-(k+1)]
                previous_positions[0] = d.pos[0]
                for l in range(0,10):
                    if previous_positions[l]==None:
                        empty += 1
                if vel > 30:
                    vel = 30
                if vel < -30:
                    vel = -30
                #set_trace()
                if vel > 0:
                    friction_dir=-1
                    d.pos[0] += 1
                    d.pos[1] = y[d.pos[0]]
                    if d.gradients[1][1]>0:
                        vel -= np.rint(5*(9.81*np.sin(d.angles[0])*np.cos(d.angles[0]) + friction_dir*d.mu_k*9.81*(np.cos(d.
                    if d.gradients[1][1]<0:
                        vel += np.rint(5*(9.81*np.sin(d.angles[0])*np.cos(d.angles[0]) + friction_dir*d.mu_k*9.81*(np.cos(d.
                    vel -= 1
                    h_diff = d.gradients[0][1]
                if vel < 0:
                    friction_dir=1
                    d.pos[0] -= 1
                    d.pos[1] = y[d.pos[0]]
                    if d.gradients[1][1]>0:
                        vel += np.rint(5*(9.81*np.sin(d.angles[0])*np.cos(d.angles[0]) + friction_dir*d.mu_k*9.81*(np.cos(d.
                    if d.gradients[1][1]<0:
                        vel -= np.rint(5*(9.81*np.sin(d.angles[0])*np.cos(d.angles[0]) + friction_dir*d.mu_k*9.81*(np.cos(d.
                    vel += 1
                    h_diff = d.gradients[1][1]

                d.gradients = [[1, y[d.pos[0]+1]-y[d.pos[0]]],[-1,y[d.pos[0]-1]-y[d.pos[0]]]]
                d.angles = [np.arctan(d.gradients[0][1]/d.gradients[0][0]),np.arctan(d.gradients[1][1]/d.gradients[1][0])]
                steps = abs(vel)

                d.capacity = d.e_const*d.mass*d.speed
                if d.load <= d.capacity or h_diff < 0:
                    d.load += d.capacity*min((d.capacity-d.load)*d.p_erosion,(-h_diff))
                    if vel > 0:
                        output[d.pos[0]-1] -= e_custom*d.capacity*min((d.capacity-d.load)*d.p_erosion,(-h_diff))
                    elif vel < 0:
                        output[d.pos[0]+1] -= e_custom*d.capacity*min((d.capacity-d.load)*d.p_erosion,(-h_diff))
                elif d.load >= d.capacity or h_diff > 0:
                    d.load -= (d.load-d.capacity)*d.p_deposition
                    if vel > 0:
                        output[d.pos[0]-1] += e_custom*(d.load-d.capacity)*d.p_deposition
                    elif vel < 0:
                        output[d.pos[0]+1] += e_custom*(d.load-d.capacity)*d.p_deposition
                it_num += 1
                if it_num == 50:
                    it_num = 0
                    break
                if (empty==0 and np.std(previous_positions)<5):
                    break
                #set_trace()

plt.figure(figsize=(256,48))
plt.plot(x, output, linewidth=10)
```

In [9]:

```
plt.figure figsize=(256,48))
plt.plot(x, y, linewidth=10)
```

In [9]:
```
plt.figure figsize=(256,48))
plt.plot(x, y, linewidth=10)
```

Out[9]:
```
[<matplotlib.lines.Line2D at 0x1d37aa6a608>]
```