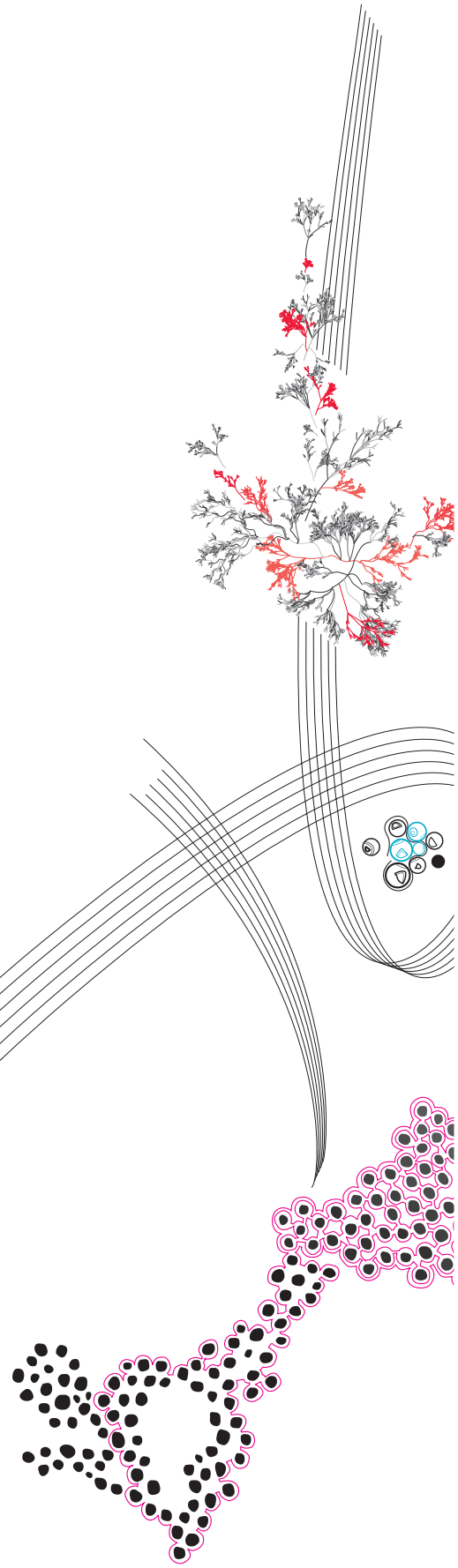

BSc Thesis Applied Mathematics

A graphical tool for constructing mathematical fractals for the AMT Toolbox

Daan Pluister

Supervisor: M.C. Boldy

June 26, 2020

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

This report is written as part of my Bachelor assignment.

Acknowledgment

I would like to thank my supervisor M.C. Boldy for help during my research and for providing feedback on the report.

Contents

A graphical tool for constructing mathematical fractals for the AMT Toolbox

Daan Pluister*

June 26, 2020

Abstract

This paper presents a design of a tool that can model mathematical fractals as part of the Toolbox for the subject Art, Mathematics and Technology (AMT). For creating fractals L-systems and Iterated Function Systems are used. This paper gives a concept for the tool and a Python implementation for the two dimensional case. For three dimensional fractals different representations of the three dimensional rotation group are studied. Also interesting for the AMT toolbox is the fractal dimension of a picture, the fractal dimension is studied and a method to find this is provided. In the end the tool is able to be used by graphical designers in such a way that the Mathematics stays hidden.

Keywords: L-systems, IFS fractals, fractal dimension, AMT Toolbox

1 Introduction

The subject Art, Mathematics and Technology (AMT) is a subject given to creative technology students. The course exists of a workshop part and practical part [2]. Not all the students are advanced with mathematics. Therefore a toolbox will be created. The toolbox should contain tools that use mathematics.

This paper focuses at one of the workshops of AMT, the self similar shapes like fractals. When you speak of fractals one property that comes to mind is the fractal dimension. The fractal dimension will be discussed. In computer graphics there are several way of building fractals, using so called L-systems or using iterated function systems.

The tool created should be able to generate 2D and 3D systems. Therefore 3D rotations are studied as well.

The research question becomes: "How can you make fractal structures accessible to graphical designers in such a way that the underlying mathematics remains hidden?"

2 Historical background

The study of fractals rose from the analysis on continuous functions and their differentiability property. The first publication of a continuous function that is nowhere differentiable is from Weierstrass [3, p 3-10]. Von Koch tried to construct a simpler example using geometry. He defined a line that is continuous but does not have a tanged anywhere and is now known as the von Koch curve [3, p 25-46]. The study of fractals was also progressed with Cantor, he constructed the Cantor set from the closed interval from 0 to 1, by removing an open interval as the middle third section recursively. A set is created with infinitely many points, while no interval is contained in it. The first few iterations of the Cantor set creation can be seen in Figure 1 [3, p 11-24].

FIGURE 1: The Cantor set, a fractal. The first seven iterations are shown.

This approach in two dimensions was experimented by Sierpinski and created the Sierpinski triangle (Figure 4). Another example is the Sierpinski carpet, which uses the same method but with a square.

When finding properties of these structures one particular aspect, the fractal dimension, comes to mind.

2.1 Fractal dimension

Unlike a line or a smooth curve a fractal can have a dimension that is not an integer. This is called the fractal dimension. The cantor set in Figure 1 for example has dimension $\log(2)/\log(3)$ (see Example 5.1). The fractal dimension can be important in art.

2.1.1 Definition fractal dimension

There are different interpretations of fractal dimension. Barnsley [1] says the following about fractal dimensions:

> They are attempts to quantify a subjective feeling which we have about how densely the fractal occupies the metric space in which it lies. Fractal dimensions provide an objective means for comparing fractals. (p. 172)

So the fractal dimension is a property of a fractal structure and thus it is useful to study. The fractal dimension can be analytically determined, but in most cases this is very cumbersome or even near impossible. The property can also be measured, a famous example is by Mandelbrot [8] on the fractal dimension of the coast of Britain. He found a dimension of 1.25, which is not an integer value.

2.1.2 Fractal dimension in art

It may come as a surprise, but some artwork can be characterised with their dimension. For example Taylor et al. [10] did a study on the fractal dimension of Jackson Pollock's drip paintings. It is found that the dimension of Pollock's paintings increased over the years.

2.2 Three dimensions

In the middle of the nineteenth century, Willam Rowan Hamilton made a walk in Dublin during which he discovered the fundamental formula for quaternion multiplication. The quaternions became the primary way of denoting vectors [5]. Later this notion was replaced by the vector analysis, because it was simpler and cleaner. So quaternions disappeared to the background. However in the late 20th century they are introduced again for their description in spatial rotations. This rotation property turns out to be very useful when creating 3D fractals.

3 Introduction to L systems

One way of describing iterated structures is using Lindenmayer-systems (L-systems). Prusinkiewicz and Lindenmayer [9] used L-systems as a way of modeling biological plants. L-systems are named after Aristid Lindemayer, he used them as a mathematical theory of plant development and are first introduced in 1968 [7].

3.1 Formal definition

For formality the definition of an L-system will be given. Before doing so, an alphabet is needed. An alphabet is a nonempty finite set, an element of an alphabet is called a character. Multiple characters in a row is called a string.

Definition 3.1. *Let V denote an* **alphabet**. *Then V^* denotes the set of all strings over V. Let λ be the empty string, a string containing no letters, and $V^+ := V^* \setminus \{\lambda\}$ the set of all nonempty strings over V.*

For example if $\{\mathtt{a}, \mathtt{b}\}$ is the alphabet then `abba` is an example of a string. Continuing, a production is also needed for an L-system.

Definition 3.2. *A* **production** *is an element $V \times V^*$, written as $\mathtt{a} \to \chi$, where $\mathtt{a}$ is a character in alphabet V and χ a string in V^*. The character $\mathtt{a}$ and the string χ are called the* **predecessor** *and the* **successor** *of this production, respectively.*

The easiest production is the following:

Definition 3.3. *A production is called an* **identity production** *when the predecessor and the successor are the same.*

Now the definition of a context free L-system can be given.

Definition 3.4. *A* **string context-free L-system** *(denoted* **string OL-system***) is an ordered triplet $G = \langle V, w, P \rangle$ where V is an alphabet, $w \in V^+$ is a nonempty string called the* **axiom** *and $P \subset V \times V^*$ is a finite* **set of productions***. If no production is explicitly specified for a given predecessor $\mathtt{a} \in V$, the identity production $\mathtt{a} \to \mathtt{a}$ is assumed to belong to the set of productions P.*

A special but important type of OL-systems is:

Definition 3.5. *An OL-system is said to be* **deterministic** *(***DOL-system***) when for each $\mathtt{a} \in V$ there is exactly one $\chi \in V^*$ such that $\mathtt{a} \to \chi$.*

Most of the OL-systems used here are deterministic. A derivation of a OL-system transforms any word of the alphabet to a new word:

Definition 3.6. *Let $\mu = a_1 \dots a_m$ be an arbitrary word over V. The word $\nu = \chi_1 \dots \chi_m \in V^*$ is* **generated** *by μ, written as $\mu \implies \nu$, if and only if $a_i \to \chi_i$ for all $i = 1, \dots, m$. A word ν is generated by an OL-system in a derivation of length n if there exists a generated sequence of words $\mu_0, \mu_1, \dots, \mu_n$ such that $\mu_0 = w, \mu_n = \nu$ and $\mu_0 \implies \mu_1 \implies \cdots \implies \mu_n$.*

A simple use of DOL-systems and very relevant is the description of the von Koch snowflake. Before stating this example we have to introduce the turtle interpretation of strings.

3.2 Turtle interpretation of strings

The turtle interpretation of strings is a way of transforming a string into a figure. A turtle can be seen as an object moving in space drawing a line.

The state of a turtle is given by the tuple (x, y, α) where (x, y) represent Cartesian coordinates and α the direction where the turtle is headed. Given a step size d and angle increment δ the turtle can use the following commands:

- `F` Move forward by a step of length d. The new state becomes $(x + d\cos\alpha, y + d\sin\alpha, \alpha)$. The line between the previous state and this state is drawn.
- `f` Move forward by step of length d, but no line is drawn.
- `+` Turn in positive direction (counter clockwise) by angle δ, the new state is $(x, y, \alpha + \delta)$.
- `-` Turn in negative direction by angle δ, the new state is $(x, y, \alpha - \delta)$.

Any other character in the string do not influence the state of the turtle. Given a string ν, the initial state of the turtle and fixed parameters d and α the turtle is able to draw a set of lines in response to the string ν. For example if the initial state is $(0, 0, 0)$, $\nu =$ `F + F + F + F`, $\delta = \frac{\pi}{2}$ and $d = 1$ then the turtle interpretation is a square with vertices at $(0, 0), (1, 0), (1, 1)$ and $(0, 1)$ (Figure 2).

Even when applying this simple interpretation of DOL-systems, nice pictures can be generated. As illustrated in the following example:

Example 3.1 (Von Koch curve)**.** *Let the step length be $d = 1$ and angle increment*

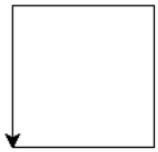

FIGURE 2: Turtle interpretation of the string `F + F + F + F` using initial state $(0, 0, 0)$, angle increment $\delta = \frac{\pi}{2}$ and step size $d = 1$.

be $\alpha = \frac{1}{3}\pi$*. Then the DOL-system*

$$w : F$$
$$p : F \to F + F - - F + F$$

generates the von Koch snowflake illustrated in Figure 3, where in each iteration the structure becomes three times bigger.

Taking the limit of these iterations, there is already the first example of a fractal generated by an L-system. Another fractal example:

Example 3.2. *The L-system given by*

$$w : F_1$$
$$p_1 : F_1 \to F_1 - F_2 + F_1 + F_2 - F_1$$
$$p_2 : F_2 \to F_2 F_2$$

generates the well known Sierpinski triangle, this can be seen in Figure 4 where the outlines of the filled triangles are generated by this L-system. Note that F_1 *and* F_2 *both have the operation of* F*.*

3.3 Branching

A tree consists of many branches, to reduce the size of a string a stack in introduced. Formally a stack is a last in first out queue, with push you can push onto the stack and

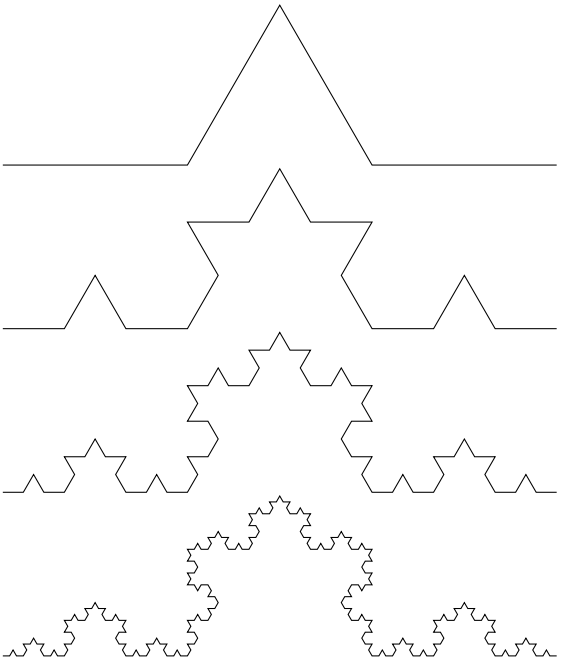

FIGURE 3: The first four iterations of the von Koch snowflake

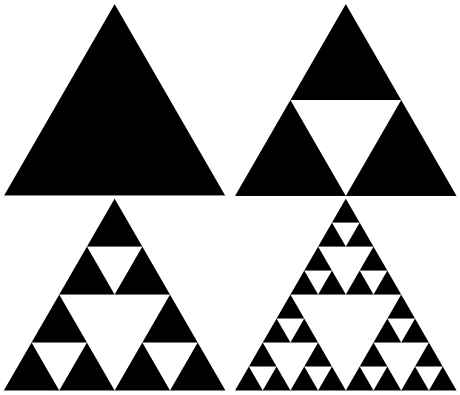

FIGURE 4: The Sierpinski triangle for $n = 1, \ldots, 4$

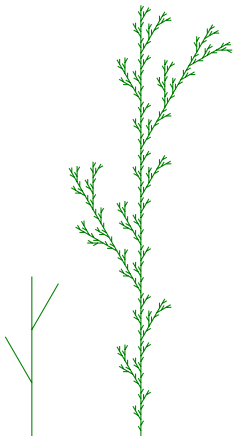

FIGURE 5: L-system of Example 3.3 with $n = 1$ (right, scaled up with 10) and $n = 4$ (left) and $\delta = pi/6$.

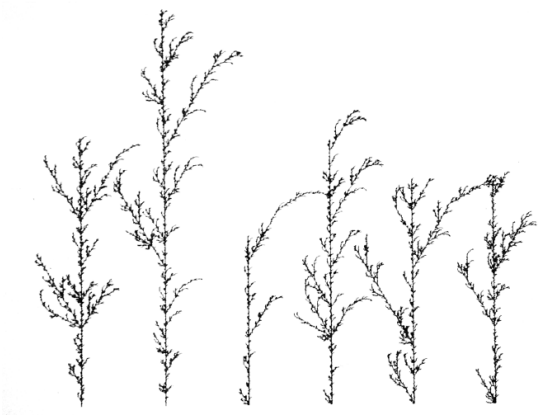

FIGURE 6: Stochastic branching structures

with pop you can get the last element that is last pushed. This means that before creating a branch the current state can be saved. Later, once the branch is drawn the turtle can immediately go back to the saved state instead of doing all the steps backwards. In an L-system the following operations should be added:

- `[` Save the current state of the turtle on a stack.
- `]` Pop the last saved state from the stack and set the turtle to this state. No line will be drawn.

Example 3.3. *The L-system with branching operations given by*

$$p : \texttt{F} -> \texttt{F[+F]F[-F]F}$$

generates Figure 5 using angle $\delta = pi/6$

3.4 Stochastic L-systems

The L-systems till now could only generate one specific figure. When probabilities are added to productions it will be uncertain that a character will generate. In this way a family strings can be constructed using the same L-system. The strings will all have the same properties but will look different. These kind of L-systems are called stochastic L-systems, where an additional dimension is added: for every production you will have a probability of it generating. To demonstrate Example 3.3 is slightly modified.

Example 3.4. *The original production is replaced with stochastic productions. So for every* `F` *there is a one in four chance that no branch will grow, only the right branch, only the left branch or all branches will grow.*

$$w : \texttt{F}$$

$$p_1 : \texttt{F} \xrightarrow{1/4} \texttt{F[+F]F[-F]F}$$

$$p_2 : \texttt{F} \xrightarrow{1/4} \texttt{F[+F]F}$$

$$p_3 : \texttt{F} \xrightarrow{1/4} \texttt{F[-F]F}$$

Using the same angle increment as in Example 3.3, a few resulting trees can be found in Figure 6. All these figures look similar but are different.

4 IFS fractals

Another way of generating fractals is using iterated function systems (IFS). An IFS is a system of contraction mappings, where a contraction mapping is an affine transformation $f(x)$ such that the distance between x_1 and x_2 is strictly greater than the distance between $f(x_1)$ and $f(x_2)$. The formal definition being:

Definition 4.1. *A transformation* $f : X \to X$ *on a metric space* (X, d) *is a* **contraction**

mapping *if there is a constant* $0 \leq s < 1$ *such that*

$$d(f(x), f(y)) \leq s \cdot d(x, y) \forall x, y \in X.$$

Any such number s *is called a* **contractivity factor** *for* f*.*

A contraction mapping has a result which is very useful for fractals: when such a mapping is iterated it converges to a single point. Formally:

Theorem 4.1 (Contraction Mapping Theorem)**.** *Let* $f : X \to X$ *be a contraction mapping on a complete metric space* (X, d)*. Then* f *possesses exactly one fixed point* $x_f \in X$ *and moreover for any point* $x \in X$*,*

$$\lim_{n \to \infty} f^{\circ n}(x) = x_f, \text{ for each } x \in X.$$

The proof is given by Barnsley [1, p. 76]. In principle an IFS is a set of these contraction mappings:

Definition 4.2. *An* **iterated function system (IFS)** *is a a finite set of contraction mappings* $w_n : X \to X$*, with respective contractivity factors* s_n*, for* $n = 1, 2, \ldots, N$*. The IFS is notated as* $\{X : w_n, n = 1, 2, \ldots, N\}$ *and its contractivity factor is* $s = \text{Max}\{s_n : n = 1, 2, \ldots, N\}$*.*

The contractivity factor follows from the fact that when the combined result of multiple contraction mappings is applied on a set of points, the contractivity factor becomes $s = \text{Max}\{s_n : n = 1, 2, \ldots, N\}$. A proof for this can be found in Barnsley [1, p. 81].

For an IFS $\{X : w_n, n = 1, 2, \ldots, N\}$ it can be shown that

$$W(B) = \bigcup_{n=1}^{N} w_n(B)$$

is a contraction mapping mapping on the space of compact subsets [1, p. 82]. As a contraction mapping converges to a fixed point and an IFS can be described with a contraction mapping on space of complete subsets, an IFS converges to a complete subset:

Definition 4.3. *The fixed set of points*

$$A = \lim_{n \to \infty} W^{\circ n}(B), \text{ for any set } B$$

is called the **attractor** *of the IFS.*

Essentially the attractor is the fractal where the IFS converges to. So IFS fractal will refer to the attractor of the IFS.

The Sierpinski triangle from Figure 4 can be defined as an IFS fractal:

Example 4.1. *The IFS of the form* $\{\mathbb{R}^2; w_1, w_2, w_3\}$ *with*

$$w_1 \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$
$$w_2 \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 0 \\ 0.5 \end{pmatrix}$$
$$w_3 \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$$

This may look complicated but in fact it is just three affine transformations copying the original structure to three different locations. However, the notation is cumbersome, therefore an IFS of the form $\{\mathbb{R}^2; w_i, i = 1, \ldots, N\}$ can be written as

$$w_i \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} a_i & b_i \\ d_i & c_i \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} e_i \\ f_i \end{pmatrix}$$

Then Table 1 represents the same system. This description of an IFS is called the IFS code. The p_i values in the IFS code determines how often w_i is applied with respect to the other p_i's. The sum of the p_i's must be one, as the p_i's are probabilities. For the Sierpinski triangle these p values are the same and thus it is a deterministic IFS. In Example 4.2 distinct values for p_i are given, which means that the density of the contraction mappings should differ to get a decent result. This example thus needs the random IFS algorithm.

Example 4.2 (The Barnsley Fern)**.** *The IFS code in Table 2 will generate the picture in Figure 7.*

5 Fractal dimension

The fractal dimension says something about how the detail changes with scale.

Definition 5.1. *Let ε the scaling factor and D the fractal dimension. Then*

$$N_\varepsilon = \varepsilon^{-D}, \tag{1}$$

can be used to calculate the number of blocks with length $1/2^\varepsilon$ needed to cover an attractor, N_ε.

By rewriting Eq 1 the fractal dimension can be found with

$$D = \lim_{\varepsilon \to 0^+} \frac{\log(N_\varepsilon)}{\log(1/\varepsilon)}.$$

Example 5.1. *The cantor set in Figure 1 leaves two parts of the interval when scaled with $1/3$, thus $N_{1/3} = 2$, in general $N_{1/3^n} = 2^n$. The fractal dimension becomes*

$$D = \frac{log(2)}{log(3)} \approx 0.63.$$

However the fractal dimension only says something about the structure in the limit, because otherwise this is not a fractal.

Example 5.2. *The von Koch curve Figure 3 has $N_{1/3} = 4$ parts added per stick, in general $N_{1/3^n} = 4^n$, so the dimension is*

$$D = \frac{log(4)}{log(3)} \approx 1.26.$$

The calculation can not always be done in this structured way. Since it may not be easy to find the increased detail. Therefore the following result is needed:

TABLE 1: IFS code for a Sierpinski triangle

w	a	b	c	d	e	f	p
1	0.5	0	0	0.5	0	0	1/3
2	0.5	0	0	0.5	0	0.5	1/3
3	0.5	0	0	0.5	0.5	0.5	1/3

TABLE 2: IFS code for the Barnsley fern

w	a	b	c	d	e	f	p
1	0	0	0	0.16	0	0	0.01
2	0.85	0.04	-0.04	0.85	0	1.6	0.85
3	0.2	-0.26	0.23	0.22	0	1.6	0.07
4	-0.15	0.28	0.26	0.24	0	0.44	0.07

Theorem 5.1 (Box counting method)**.** *Let A be an attractor. Cover $\mathbb{R}^m$ by closed just-touching square boxes of side length $(1/2)^n$. Let $N_n(A)$ be the number of boxes of side length $(1/2)^2$ which intersect the attractor A. If*

$$D = \lim_{n \to \infty} \frac{\log(N_n(A))}{\log(2^n)}$$

then A has fractal dimension D.

The box counting method complies with the dimension:

Example 5.3. *Fractal dimension of a square can be determined by Theorem 5.1. $N_1 = 4, N_2 = 16$ and in general $N_n = 4^n$. Therefore*

$$D = \lim_{n \to \infty} \frac{\log(4^n)}{\log(2^n)} = 2.$$

This is the desired result.

The box counting method makes it also easy for finding the dimension of the Sierpinski triangle

Example 5.4. *The Sierpinski triangle from Table 1*

$$D = \lim_{n \to \infty} \frac{\log(3^n)}{\log(2^n)} = 1.58.$$

6 3D fractals

To extend the fractals made with L-systems and IFSs to 3D, various representations will be explored. For L-systems in particular a search for a well 3D turtle interpretation. The 2D turtle from subsection 3.2 can now be viewed as if it is swimming. So the turtle can also yaw, pitch and roll.

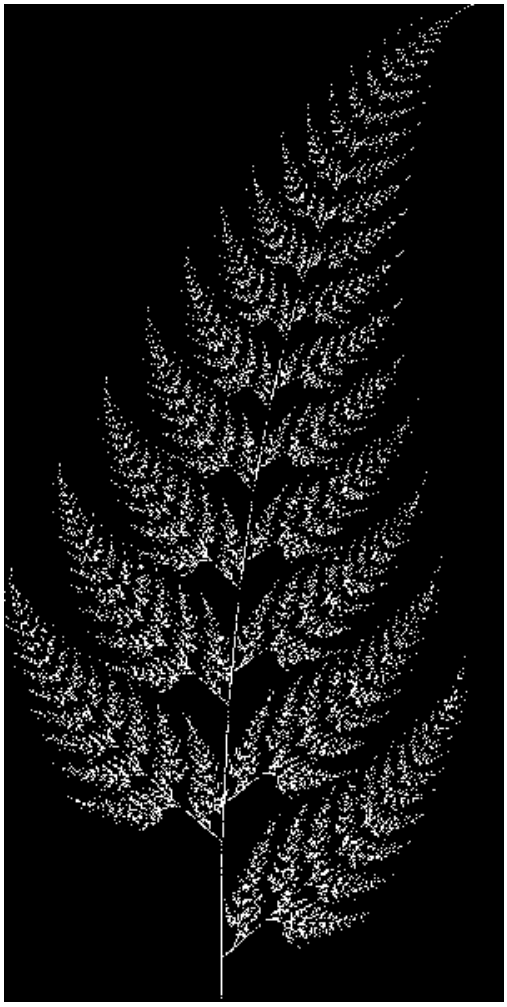

FIGURE 7: Barnsley fern generated with the IFS code of Figure 7 with $n = 36$

6.1 Euler angles

Prusinkiewicz and Lindenmayer [9] uses the common Euler angles to describe a rotation of the turtle in 3D.

The orientation of the turtle is now defined by three vectors. A vector for the heading direction of the turtle $\vec{H}$, a vector on the left hand of the turtle $\vec{L}$ and a vector up $\vec{U}$. These vectors are perpendicular and thus satisfy $\vec{H} \times \vec{L} = \vec{U}$. A rotation can be described as a rotation around these vectors. $\vec{H}$, used for a roll rotation, $\vec{L}$ for a pitch rotation and $\vec{U}$ for a yaw rotation. Rotations are expressed by the equation

$$\begin{pmatrix} \vec{H} & \vec{L} & \vec{U} \end{pmatrix} \mathbf{R} = \begin{pmatrix} \vec{H}' & \vec{L}' & \vec{U}' \end{pmatrix}$$

where $\mathbf{R}$ is a 3×3 rotation matrix. Specifically, rotations by an angle α about the vectors $\vec{H}, \vec{L}$ and $\vec{U}$ are represented by the matrices:

$$\mathbf{R_U}(\alpha) = \begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (2)$$

$$\mathbf{R_L}(\alpha) = \begin{pmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{pmatrix} \quad (3)$$

$$\mathbf{R_H}(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \quad (4)$$

The following operations control the turtle in space (as in Figure 8)

- `+` Yaw left by angle δ, using rotation matrix $\mathbf{R_U}(\delta)$.
- `-` Yaw right by angle δ, using rotation matrix $\mathbf{R_U}(-\delta)$.
- `&` Pitch down by angle δ, using rotation matrix $\mathbf{R_L}(\delta)$.
- $\wedge$ Pitch up by angle δ, using rotation matrix $\mathbf{R_L}(-\delta)$.
- `\` Roll left by angle δ, using rotation matrix $\mathbf{R_H}(\delta)$.
- `/` Roll right by angle δ, using rotation matrix $\mathbf{R_H}(-\delta)$.
- `|` Turn around, using rotation matrix $\mathbf{R_H}(\pi)$.

6.1.1 Problems with Euler angles

The uniqueness of Euler angles will be lost when one axis is aligned with one other axis. This can be demonstrated using gimballs and two gimbals in the same plane is called a gimbal lock. Such a gimbal lock can be created with the matrices from Eq 4, Eq 3 and Eq 2 with using a roll angle of $\pi/2$. The rotation $\mathbf{R_H}(\alpha)\mathbf{R_L}(\pi/2)\mathbf{R_U}(\beta)$ created will result in the matrix in Eq 5. Using matrix multiplications the result is Eq 6. Using the

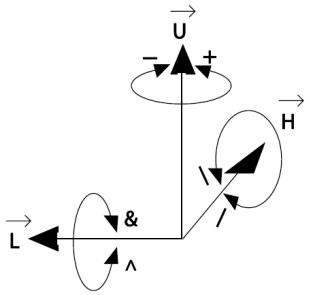

FIGURE 8: Turtle interpretation in three dimensions of Prusinkiewicz and Lindenmayer [9]

trigonometry formulas

$$\mathbf{R} = \begin{pmatrix} 0 & 0 & -1 \\ -\sin(\alpha+\beta) & \cos(\alpha+\beta) & 0 \\ \cos(\alpha+\beta) & \sin(\alpha+\beta) & 0 \end{pmatrix}$$

becomes the resulting matrix. Then for this rotation it does not matter if you change α or β since these rotate around the same axis. Thus after using matrix R_L there should be two degrees of freedom, but one degree of freedom is lost.

6.2 Axis angle

A more intuitive way of rotating a body in space is using a rotation around any vector. By Euler's rotation theorem can be achieved:

Theorem 6.1 (Euler's rotation theorem)**.** *Any rotation can be described by an direction vector and an angle.*

So given a unit vector $\mathbf{v}$ in $\mathbb{R}^3$ and an angle θ, then let $(\mathbf{v}, \theta)$ represent the positive (counter clockwise) rotation around the axis defined by the unit vector looking from the origin. This positive direction can be seen using the right hand rule trick. When you point your thumb in the direction of the unit vector and curling the rest of your fingers around this vector, then these fingers point in the direction of the rotation.

6.2.1 Problems with Axis angle

The rotation $(\mathbf{v}, \theta)$ and $(-\mathbf{v}, -\theta)$ for all $\mathbf{v}$ and θ correspond to the same rotation. Also $(\mathbf{0}, \theta)$ is the identity rotation for any θ. Therefore you have a double representation for the same rotation. Furthermore the composition of two rotations can not be easily found in this notation.

6.3 Quaternions

A rotation with an angle of θ around the axis trough a vector $u = (u_x, u_y, u_z)$ can be represented by a quaternion of unit length. A good introduction of quaternions can be found in Kuipers [6].

Theorem 6.2. *Let* $\mathbf{u} = (u_x, u_y, u_z)$ *be a vector of unit length and* $\mathbf{q}$ *the quaternion*

$$\begin{aligned} \mathbf{q} &= e^{\frac{\theta}{2}(u_x \boldsymbol{i} + u_y \boldsymbol{j} + u_z \boldsymbol{k})} \\ &= \cos\frac{\theta}{2} + (u_x \boldsymbol{i} + u_y \boldsymbol{j} + u_z \boldsymbol{k}) \sin\frac{\theta}{2}, \end{aligned} \tag{7}$$

then the quaternion multiplication

$$\mathbf{p}' = \mathbf{q}\mathbf{p}\mathbf{q}^{-1} \tag{8}$$

can be used to rotate $\mathbf{p}$ *with* θ *around the axis spanned by* $\mathbf{u}$*.*

The operation in (8) is, in terms of group theory, the conjugation of $\mathbf{p}$ by $\mathbf{q}$. Where $\mathbf{p}^{-1}$ is the inverse of $\mathbf{p}$ given by

$$\begin{aligned} \mathbf{q}^{-1} &= e^{-\frac{\theta}{2}(u_x \mathbf{i} + u_y \mathbf{j} + u_z \mathbf{k})} \\ &= \cos\frac{\theta}{2} - (u_x \mathbf{i} + u_y \mathbf{j} + u_z \mathbf{k}) \sin\frac{\theta}{2}. \end{aligned}$$

Since $\mathbf{q}$ is of unit length the inverse is the same as the quaternion conjugate $\mathbf{q}^*$.

6.3.1 Extending the turtle to 3D

Representing the quaternion in Eq 7 as a tuple (a, b, c, d) where

$$\begin{aligned} a &= \cos\frac{\theta}{2} \\ b &= u_x \sin\frac{\theta}{2} \\ c &= u_y \sin\frac{\theta}{2} \\ d &= u_z \sin\frac{\theta}{2} \end{aligned}$$

,

$$\mathbf{R} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} \cos(\beta) & \sin(\beta) & 0 \\ -\sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5}$$

$$\mathbf{R} = \begin{pmatrix} 0 & 0 & -1 \\ \sin(\alpha)(-\cos(\beta)) - \cos(\alpha)\sin(\beta) & \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta) & 0 \\ \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta) & \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta) & 0 \end{pmatrix} \tag{6}$$

then using the calculation

$$\begin{pmatrix} \vec{H}' & \vec{L}' & \vec{U}' \end{pmatrix} = \mathbf{q} \begin{pmatrix} \vec{H} & \vec{L} & \vec{U} \end{pmatrix} \mathbf{q}^{-1}$$

Since the imaginary part is made out of a unit vector times $\sin(\theta/2)$ the imaginary part can be noted as the vector $\vec{u}$ times this sine. So $(a, b, c, d) = (a, \vec{u}\sin\frac{\theta}{2})$. Then the string operations in subsection 6.1 can be defined as

- $+$ Yaw left by angle δ, using the quaternion $\mathbf{q} = (\cos\frac{\delta}{2}, \sin\frac{\delta}{2}\vec{U})$.
- $-$ Yaw right by angle δ, using the quaternion $\mathbf{q} = (\cos\frac{-\delta}{2}, \sin\frac{-\delta}{2}\vec{U})$.
- & Pitch down by angle δ, using the quaternion $\mathbf{q} = (\cos\frac{\delta}{2}, \sin\frac{\delta}{2}\vec{L})$.
- $\wedge$ Pitch up by angle δ, using the quaternion $\mathbf{q} = (\cos\frac{-\delta}{2}, \sin\frac{-\delta}{2}\vec{L})$.
- $\backslash$ Roll left by angle δ, using the quaternion $\mathbf{q} = (\cos\frac{-\delta}{2}, \sin\frac{-\delta}{2}\vec{H})$.
- / Roll right by angle δ, using the quaternion $\mathbf{q} = (\cos\frac{\delta}{2}, \sin\frac{\delta}{2}\vec{H})$.
- | Turn around, using the quaternion $\mathbf{q} = (\cos\frac{\pi}{2}, \sin\frac{\pi}{2}\vec{U})$.

6.3.2 Problems with quaternions

The uniqueness of the quaternions representation is lost in the same way the axis angle representations loses this. Namely the rotation by (a, b, c, d) is the same rotation as $(-a, -b, -c, -d)$. However in contrast to the axis angle representation the composition of two rotations $\mathbf{q_1}$ followed by $\mathbf{q_2}$ is easily found and is given with:

$$\mathbf{q} = \mathbf{q_2}\mathbf{q_1}.$$

7 Programming

Based on the theory in the previous sections a tool can be created. The main use cases should be:

- Process of a parametric L-system and the visualization of the resulting string.
- The ability to generate fractals from iterated function systems from an IFS code.
- Box counting method.

7.1 L-systems

The L-system implementation is build with object oriented programming. A class *LSystem* is made which has the same properties as how a L-system is defined. So it has an axiom and productions. The method *iterate(n)* processes the axiom n times and updates the axiom in the *LSystem* object according to the set of productions. The method *drawLsystem(turtle)* draws the axiom of the *LSystem* object with the given *turtle*. The turtle is defined using the *Turtle graphics* package [4]. The implementation in subsection A.1 is able to create simple L-systems. The branching and stochastic implementations should be easy to implement. Once the three dimensional case is implemented using the quaternions, the tool should be able to generate the L-system in Example 7.1.

Example 7.1. *Using an angle $\delta = \pi/8$ and*

$n = 7$ *the system*

w : `A`

p_1 : `A` $\rightarrow$ `[&FL!A]/////'[&FL!A]`
`///////'[&FL!A]`

p_2 : `F` $\rightarrow$ `S /////ˆ F`

p_3 : `S` $\rightarrow$ `F L`

p_4 : `L` $\rightarrow$ `['''`$\wedge\wedge${`-f+f+f-|-f+f+f}]`

generates Figure 9.

FIGURE 9: The tree generated by Example 7.1

7.1.1 Two dimensional Python implementation

For the two dimensional case a basic python implementation of algorithm 2 is given in subsection A.1.

7.2 IFS

As described by Barnsley [1] a deterministic algorithm for generating the attractor of an IFS $\{\mathbf{X}; w_1, w_2, \ldots, w_m\}$ is given in algorithm 1. By Definition 4.3 the sequence A_n in algorithm 1 converges to the attractor of the IFS.

For random IFS codes the random IFS algorithm is needed. This algorithm is similar to algorithm 1, but takes probabilities in consideration. As $n \to \infty$ the sequence x_n converges to the attractor of the IFS [1, p. 356]. Giving a deterministic IFS code to the random IFS algorithm, then for n large

Algorithm 1: Deterministic IFS algorithm

Result: For a given a number of iterations n and IFS code, the algorithm computes $A_n = W^{\circ n}(A)$.

Let $\{\mathbf{X}; w_1, w_2, \ldots, w_m\}$ be a IFS;
Choose a compact set $A_0 \subset \mathbb{R}^2$.
for $i = \{1, \ldots, n\}$ **do**
  Compute $A_{i+1} = \bigcup_{j=1}^{N} w_j(A_i)$;
end

Algorithm 2: Random IFS algorithm

Result: For a given a number of iterations n and IFS code, the algorithm computes a set of points A

Let $\{\mathbf{X}; w_1, w_2, \ldots, w_m\}$ be a IFS;
Choose a point $x_0 \in \mathbf{X}$ and then recusively, independently choose a point

$$x_n \in \{w_1(x_{n-1}), \ldots, w_m(x_{n-1})\}$$

for $n = 1, 2, \ldots$, where the probability of the event $x_n = w_i(x_{n-1})$ is p_i. Thus construct a sequence of points x_n

enough the results look the same, as they both converge to the attractor of the IFS.

7.2.1 Two dimensional Python implementation

For the two dimensional case a basic python implementation of algorithm 2 is given in subsection A.2. The implementation accepts an IFS code and generates the set A_n for a given n.

7.3 The three dimensional case

Once the quaternion representation from section 6 is implemented, the two dimensional cases can be extended to three dimensions. The L-systems by implementing the commands discussed and the IFS algorithm with a 3D representation using the same methods for mapping.

7.4 Box counting method

The box counting method can be implemented using Theorem 5.1.

8 Conclusion

The tool described in this paper will hide the mathematics. Using examples given in this paper, a graphic designer should be able to create some examples as well.

9 Discussion

An useful extension to the IFS tool could be an implementation of the *Collage theorem* by Barnsley [1]. The theorem states that an IFS can approximate self similar sets by finding the correct contraction mappings.

References

[1] Barnsley, M. (1989). Fractals Everywhere. *American Journal of Physics*, 57(11):1053–1053.

[2] Boldy, M. (2020). OSIRIS - Onderwijsaanbod 201800233 2019.

[3] Edgar, G. A. (2019). *Classics On Fractals*.

[4] Feurzig, W. and Papert, S. (2020). Turtle graphics — Python 3.3.7 documentation.

[5] Kuipers, J. B. (1999a). Historical Matters. In *Quaternions and Rotation Sequences*, pages 3–12. Princeton University Press.

[6] Kuipers, J. B. (1999b). *Quaternions and Rotation Sequences*. Princeton University Press.

[7] Lindenmayer, A. (1968). Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299.

[8] Mandelbrot, B. (1967). How long is the coast of Britain? Statistical self-similarity and fractional dimension. *Science*, 156(3775):636–638.

[9] Prusinkiewicz, P. and Lindenmayer, A. (1990). *The Algorithmic Beauty of Plants*. The Virtual Laboratory. Springer New York, New York, NY.

[10] Taylor, R. P., Micolich, A. P., and Jonas, D. (1999). Fractal analysis of Pollock's drip paintings [6].

A Python implementations

A.1 L-system

The following listing can be found at git.snt.utwente.nl/s1959190/bachelor-assignment

```
import turtle


class Production:

    def __init__(self, predecessor, successor):
        self.predecessor = predecessor
        self.successor = successor

    def __str__(self):
        return self.predecessor + " -> " + self.successor


class Operation:

    def __init__(self, symbol, turtleCommand, description=""):
        """
        :type symbol: str
        :type turtleCommand: str
        :type description: str
        """
        self.symbol = symbol
        self.turtleCommand = turtleCommand
        self.description = description

    def apply(self):
        eval("t." + self.turtleCommand)


class LSystem:

    def __init__(self, productions, axiom=None):
        self.productions = productions
        self.setAxiom(axiom)
        self.stringToDraw = axiom

    def processString(self, oldStr):
        newstr = ""
        for ch in oldStr:
            newstr = newstr + self.applyRules(ch)

        return newstr

    def setAxiom(self, axiom):
        self.axiom = axiom
```

```
    def iterate(self, numIterations=1):
        startString = self.axiom
        endString = ""
        for i in range(numIterations):
            endString = self.processString(startString)
            startString = endString

        self.stringToDraw = endString

        return endString

    def applyRules(self, ch):
        """
        :param ch a single char
        :returns new string according to L-system productions defined in globa
        """
        newstr = ch
        for prod in self.productions:  # Loop through rules replace when corre
            if ch == prod.predecessor:
                newstr = prod.successor

        return newstr

    def drawLsystem(self, aTurtle, angle=90,distance=1):
        for cmd in self.stringToDraw:
            if cmd == 'F':
                aTurtle.forward(distance)
            elif cmd == 'B':
                aTurtle.backward(distance)
            elif cmd == '+':
                aTurtle.left(angle)
            elif cmd == '-':
                aTurtle.right(angle)


def drawLsystem(aTurtle, instructions, angle, distance):
    for cmd in instructions:
        if cmd == 'F':
            aTurtle.forward(distance)
        elif cmd == 'B':
            aTurtle.backward(distance)
        elif cmd == '+':
            aTurtle.left(angle)
        elif cmd == '-':
            aTurtle.right(angle)


def main():
```

```
    p1 = Production("F", "F-F++F-F")

    print(p1)

    lsystem = LSystem([p1])
    lsystem.setAxiom("F ++ F ++ F")
    stringToDraw = lsystem.iterate(3)

    print(stringToDraw)

    t = turtle.Turtle()  # create the turtle
    wn = turtle.Screen()

    t.up()
    t.back(100)
    t.down()

    t.speed(50)
    lsystem.drawLsystem(t, 60, 5)  # draw the picture
    # angle 60, segment length 5
    wn.exitonclick()


main()
```

A.2 IFS fractals

The following listing can be found at git.snt.utwente.nl/s1959190/bachelor-assignment

```
#! /usr/bin/env python
# coding: utf-8

from __future__ import division

from json import load
from PIL import Image, ImageDraw
from random import uniform


def process_file(transformations, width, height, iterations=1, outputfile='out

    probability_join = sum(t[0] for t in transformations)

    points = set([(0, 0)])

    # for each iteration
    for i in range(iterations):
        new_points = set()

        # for each point
```

```
        for point in points:

            # decide on which transformation to apply
            rnd = uniform(0, probability_join)
            p_sum = 0
            for probability, function in transformations:
                p_sum += probability
                if rnd <= p_sum:
                    new_points.add(function(*point))
                    break

        points.update(new_points)

        if i <= 4:
            continue

        # find out image limits determine scaling and translating
        min_x = min(points, key=lambda p:p[0])[0]
        max_x = max(points, key=lambda p:p[0])[0]
        min_y = min(points, key=lambda p:p[1])[1]
        max_y = max(points, key=lambda p:p[1])[1]
        p_width = max_x - min_x
        p_height = max_y - min_y

        width_scale = (width/p_width)
        height_scale = (height/p_height)
        scale = min(width_scale, height_scale)

        # create new image
        image = Image.new( 'RGB', (width, height))
        draw = ImageDraw.Draw(image)
        draw.ink = draw.getink("#008000")

        print(draw.getink)

        # plot points
        for point in points:
            x = (point[0] - min_x) * scale
            y = height - (point[1] - min_y) * scale
            draw.point((x,y))

        # save image file
        image.save( outputfile + "/" + str(i) + '.png', "PNG" )

def parse(filename):
    with open(filename) as f:
        definition = load(f)

    # check for errors
```

```
    if "width" not in definition: raise ValueError('"width" parameter missing')
    if "height" not in definition: raise ValueError('"height" parameter missing
    if "iterations" not in definition: raise ValueError('"iterations" parameter
    if "transformations" not in definition: raise ValueError('"transformations"

    def make_t_function(expression):
        return lambda x,y: eval(expression, {'x': x, 'y': y})

    definition['transformations'] = [(float(probability), make_t_function(expr
                                      probability, expression in definition['tra

    return definition


if __name__ == "__main__":

    import sys

    # if there is one argument and it's not "-"
    if len(sys.argv) > 1 and sys.argv[1] != '-':
        # process each filename in input
        for filename in sys.argv[1:]:
            result = parse(filename)
            process_file(result['transformations'], result['width'],
                         result['height'], result['iterations'],
                         filename.split('.')[0])
    else:
        # read contents from stdin
        eval( sys.stdin.read() )
        process_file( transformation, width, height, iterations)
```