Bit analysis of 2D-Spiroplots and the generation of 3D-Spiroplots

Li-Yeun Xu

Student Number: 6521495

Supervisor:
Marc van Kreveld

Master Computing Science
Utrecht University

October 12, 2023

Abstract

This study investigates the effect of varying bit precisions on the accuracy and computational efficiency of generating Spiroplots, which are a procedural dynamical system used to generate beautiful figures created by Casper van Dommelen, Marc van Kreveld and Jérôme Urhausen in the 2020 paper: "Spiroplots: a new discrete-time dynamical system to generate curve patterns"[9]. Our research determines that while higher bit precision can marginally enhance the accuracy of a Spiroplot, it does so at a notable computational cost. Additionally, enhancements to the original Spiroplot application are proposed, introducing new point-drawing order methods for situations where points overlap on identical pixel locations. Furthermore, the second half of the paper introduces an extension to the original 2D-Spiroplot concept, paving the way for the creation of 3D-Spiroplots. By overcoming the challenges of defining a rotation axis in three-dimensional space, we have crafted methods that maintain desirable properties like retaining the center of gravity. Our proposed techniques result in intricate and symmetrical 3D patterns that are both mathematically consistent and visually appealing. This paper is accompanied by an application that allows for the running of 2D and 3D-Spiroplots.

Contents

1 Introduction **2**

2 Related works **3**

2.1 Mechanical constructions 3
2.2 Parameterized curve 3
2.3 Lissajous figures 4
2.3.1 Spirographs 4
2.3.2 Rose Curves 6
2.4 Algorithmic drawing techniques 7
2.5 Grammar-based Systems 9
2.5.1 L-systems 9
2.5.2 Context-free Design Grammar 9
2.5.3 Shape Grammar 10
2.6 Iterated Function Systems 11
2.7 Cellular automata 13
2.8 Coupled equation systems 13
2.8.1 Hénon map 14
2.8.2 Lorenz system 15
2.9 Agent Based Methods 15
2.10 Particle systems 16
2.11 Artificial neural networks 17

3 2D-Spiroplots **17**
3.1 Bit analysis of Spiroplots 19
3.1.1 Visual quality analysis 22
3.1.2 Quantitative error analysis 24
3.1.3 Computational Efficiency 32
3.2 Enhancements to the original Spiroplot application 34

4 3D-Spiroplots **35**
4.1 Rotation methods 36
4.1.1 Vector from the center of mass to the midpoint as the directed axis 36
4.1.2 Axis parallel to one of the position vectors 37
4.1.3 Multiple position vectors axis rotation 37
4.1.4 Normal vector of a plane as the directed axis 41
4.2 Properties of 3D-Spiroplots 45
4.3 3D-Spiroplot application 48

5 Conclusion and Discussions **48**

Appendices **50**

A Image comparisons **50**

1 Introduction

Processes that lead to beautiful figures exist in various forms. When these processes are used to create art, they are referred to as generative art. Over the past decade, the creation of such art has evolved, transitioning from mechanical tools, to using artificial intelligence. Among the diverse methods that have emerged, Spiroplots represent a novel and intriguing approach to generating captivating patterns. Their core principle revolves around a systematic arrangement of rotating pairs of points.

A significant portion of this paper delves into the intricacies of Spiroplots, especially the impact of different bit-precision levels. Through thorough analysis, we understand how bit-precision influences the accuracy and visual appeal of the Spiroplot designs.

In the second half of the paper we extend Spiroplots into 3D. This extension introduces us to various rotation methods, with each method offering a unique set of 3D patterns. Building on the established principles of 2D-Spiroplots, the 3D versions present additional complexities.

2 Related works

This paper is largely based on the 2020 paper: "Spiroplots: a new discrete-time dynamical system to generate curve patterns"[9]. Aside from introducing Spiroplots, this foundational work also touches on other procedural systems capable of producing patterns. In this paper's related work section we delve deeper into this aspect, building upon the research presented.

2.1 Mechanical constructions

Prior to existence of computers, generative art already existed and was created through mechanical constructions. These constructions often use various mechanical components, such as gears, pulleys, motors, or pendulums, to generate visually intricate designs. Many of these constructions have since been digitally reproduced through mathematical formulas. Some notable mechanical contraptions used for making generative art are the following:

- **Pendulums**: Weight suspension mechanism that consist of a weight which gets suspended from a fix point. Under the influence of gravity a pendulum swings back and forth and creates oscillating patterns.
- **Harmonographs** [44]: Mechanical devices that use pendulums to create Lissajous curves or other geometric patterns.
- **Spirographs** [13]: Drawing toy from 1965 that consists of plastic gears with one gear inside the other. By placing a pen in one of the holes in the smaller gear and moving it along the path created by the larger gear, beautiful shapes and patterns can be drawn.

2.2 Parameterized curve

A parameterized curve is a mathematical concept that describes a smooth and continuous path through space. In generative art, parameterized curves are used to describe beautiful shapes and patterns in a compact and intuitive way.

A parameterized curve is defined by a set of mathematical functions that describe its position in space as a function of one or more parameters. For example, a simple parameterized curve might be defined by a function that describes the x-coordinate of a point on the curve as a function of a parameter t, and another function that describes the y-coordinate of the same point. By adjusting the parameters of the curve, an artist can control the shape and path of the curve, making it possible to create a wide variety of shapes and paths with relative ease. Several popular parameterized curves used in generative art are Lissajous figures, Spirographs and Rose curves.

2.3 Lissajous figures

Lissajous figures originated from a mechanical construction discovered by Jules-Antoine Lissajous in 1857 [23]. Lissajous figures are patterns created by the intersection of two simple harmonic motions, usually perpendicular to each other. In generative art Lissajous figures have been translated to Lissajous curves [29], which are a class of parameterized curves that result from the intersection of two sinusoidal functions, one in each of the x and y dimensions. A Lissajous curve is defined by the following parametric equations:

$$x(t) = A * \sin(a * t + \delta)$$

$$y(t) = B * \sin(b * t)$$

Where:

- A and B are the amplitudes in the x and y dimensions, respectively.
- a and b are the frequencies in the x and y dimensions, respectively.
- δ is the phase difference between the two sinusoidal functions.
- t is a parameter that varies within a suitable range.

By changing the values of A, B, a, b, and δ, artists can create a variety of Lissajous curve patterns with different shapes and complexities. Some examples are shown in Figure 1.

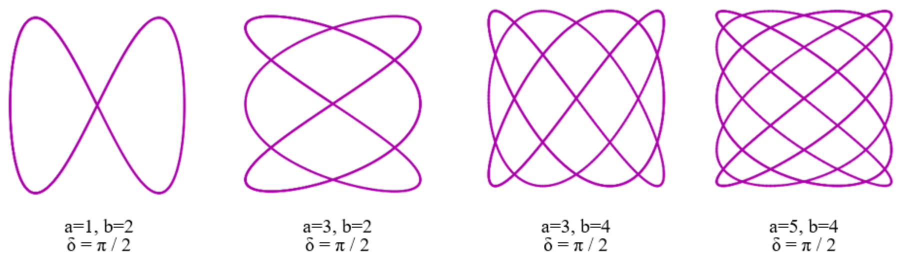


Figure 1: Lissajous curves with different parameter values [24].

2.3.1 Spirographs

In section 2.1 it is mentioned that Spirographs originated from a mechanical construction. Spirographs were invented by Denys Fisher in 1965 as a drawing toy [13]. In generative art Spirographs have been digitally reconstructed by rolling a circle inside or outside another circle while tracing a point on the rolling circle's surface. Depending on whether the rolling circle rolls inside or outside the fixed circle, the drawing becomes either a hypotrochoid or epitrochoid [20].

A hypotrochoid [22, 40] is a curve traced by a point attached to a circle of radius r rolling around the inside of a fixed circle of radius R, where the point is at a distance d from the center of the inner circle. The parametric equations for a hypotrochoid are:

$$x(t) = (R + r) * \cos(t) - d * \cos\left(\frac{R + r}{r} * t\right)$$

$$y(t) = (R + r) * \sin(t) - d * \sin\left(\frac{R + r}{r} * t\right)$$

where t is a parameter that varies within a specified range, typically from 0 to some multiple of 2π.

If the rolling circle rolls outside the fixed circle, the curve becomes a epitrochoid [22, 39] and the equations change to:

$$x(t) = (R - r) * \cos(t) + d * \cos\left(\frac{R - r}{r} * t\right)$$

$$y(t) = (R - r) * \sin(t) - d * \sin\left(\frac{R - r}{r} * t\right)$$

Figure 2 shows that by adjusting the values of R, r, and d, different Spirograph patterns can be generated.

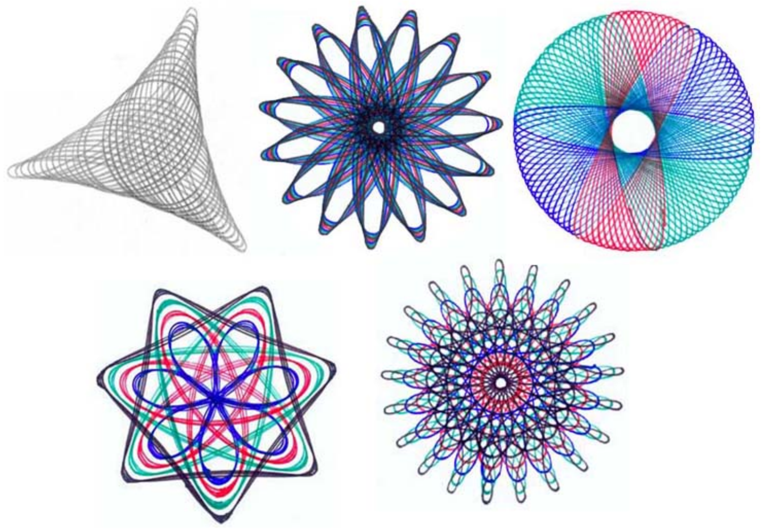


Figure 2: Spirographs with different parameter values [33].

Comparing Lissajous curves to Spirographs we observe that they produce very similar curves. However in Lissajous curves you can notice that these curves have been enclosed in a rectangular shape, whereas Spirographs are enclosed by a circular boundary.

2.3.2 Rose Curves

Rose curves [26, 41], also known as Rhodonea curves, are curves generated by plotting the polar coordinates of points on a circle of a sinusoid function. Rose curves are composed of petals, which are formed by the variation of the angle θ.

In Cartesian coordinates rose curves are defined by the following parametric equations:

$$x(\theta) = A * \cos(k * \theta) * \cos(\theta)$$
$$y(\theta) = A * \cos(k * \theta) * \sin(\theta)$$

where:

- A is a constant that determines the size of the curve.
- k is an integer that determines the number of petals on the curve.
- θ is a parameter that varies within a suitable range.

When k is non-zero, the curve will be rose-shaped with $2k$ petals if k is even, and k petals when k is odd. Figure 3 shows examples of different Rose curves.

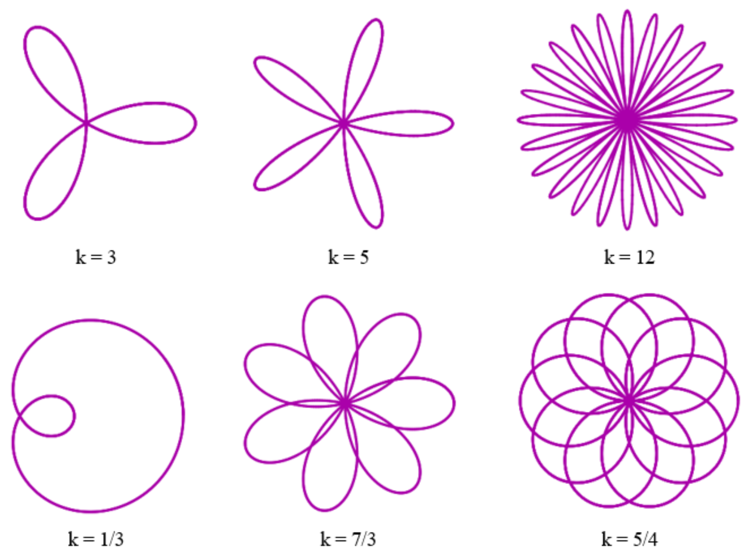


Figure 3: Rose curves with different k values [24].

2.4 Algorithmic drawing techniques

Algorithmic drawing techniques involve using simple rules and commands to create drawings or geometric patterns. A simple algorithmic drawing technique is reflection symmetry where an object or pattern gets mirrored across a line in 2D, or a plane in 3D.

Tiling is another well-known algorithmic drawing technique. The tiles used in tiling are usually identical and similar in shape, and are arranged in a way that cover a surface without gaps or overlaps.

Spirolaterals [21] are geometric patterns created by recursively connecting points on a series of lines or curves. Starting with a simple shape, such as a line or a square, new lines or curves are added at specific intervals, and the resulting points are connected to create more complex patterns. A spirolateral is defined by three factors: the turning angle, the number of segments or turns, and the number of repetitions, which create a closed figure. An example is shown in Figure 4.

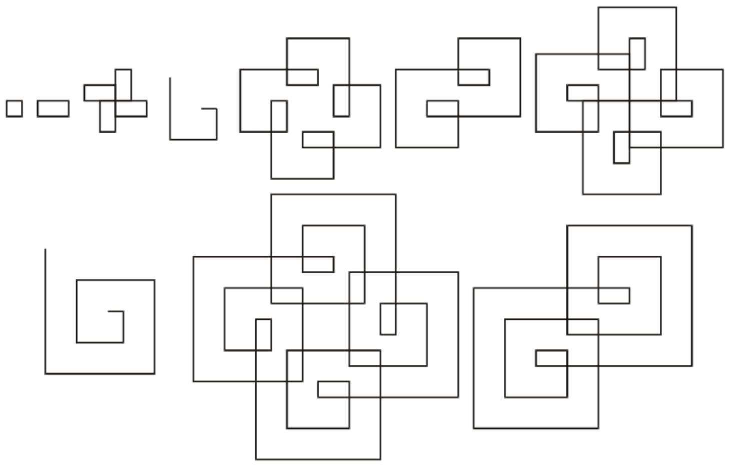


Figure 4: Spirolateral of 90 degrees, from 1 to 10 turn [21].

Turtle graphics [32], used in the Logo programming language, is also an example of an algorithmic drawing technique. In turtle graphics, an artist can control a virtual ”turtle” that moves on a canvas, leaving a trail as it moves. By issuing commands to control the turtle’s movement and turning, the artist can create geometric patterns and shapes. Visually appealing patterns and designs can be created by combining simple commands in creative ways, often using loops, recursion or randomization.

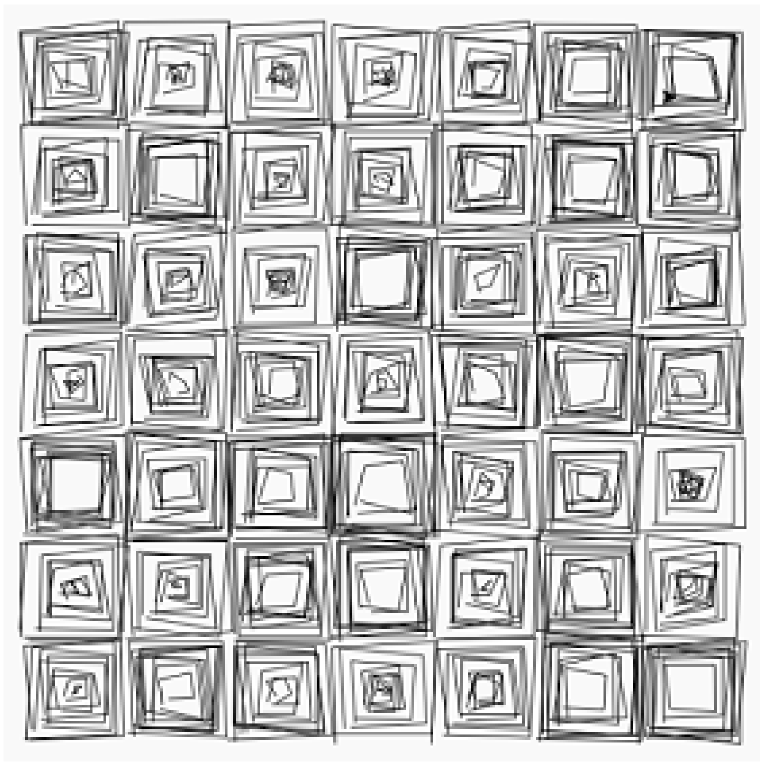


Figure 5: Vera Molnár's Artwork recreated using Turtle graphics [38].

2.5 Grammar-based Systems

Another rule-based method to create intricate and complex shapes and patterns are grammar-based methods [28]. Grammar-based methods refers to a family of techniques that use formal grammars [19] to generate complex structures or patterns. Formal grammars are defined by a set of production rules that describe how to generate sequences or structures from a given set of symbols or elements. The production rules in grammar-based methods can be applied sequentially or simultaneously and can be context-sensitive or context-free, depending on the specific grammar. In generative art these methods provide artists with a systematic and flexible way to explore a wide range of possibilities by defining and manipulating the production rules. Some popular grammar-based methods used in generative art are: L-systems [31], context-free Design grammars [6] and Shape grammars [15].

2.5.1 L-systems

L-systems [31] were introduced by Aristid Lindenmayer to model the growth of plants and other organisms, but their use has also extended to generative art. Another defining characteristic of L-systems is that they are based on parallel replacement rules, where all the production rules are applied simultaneously to all symbols or elements in the string during each iteration. An example of a L-system is the following:

$$A \rightarrow B - A - B$$

$$B \rightarrow A + B + A$$

Starting with the symbol 'A', this system would yield the following expansion: $A, B - A - B, A + B + A - B - A - B - A + B + A, \ldots,$ and so forth. The way L-system traditionally are used in generative art is by applying geometric semantics to the output. For instance, we could interpret A and B as moving one step forward, and + and − as turning 60 degrees clockwise or counter-clockwise. With this interpretation we get the output shown in Figure 6.

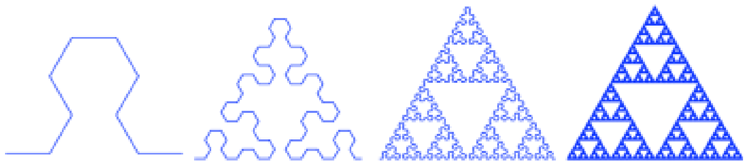


Figure 6: Sierpinski Triangle generation process by an L-system [17].

2.5.2 Context-free Design Grammar

In L-systems the symbol generation and the geometrical interpretation are two independent steps. In context-free Design grammar [6] geometry is embedded inside the production rule, by extending the syntax of formal grammars with transformation operators. The transformation operators modify the current rendering state. These operators can for example change the rotation and scaling of the current coordinate system and can make modification of the hue or saturation of the current drawing color. An example of a context-Free Design grammar is the following (Figure 7):

```
startshape SEED

rule SEED {
    SQUARE {}
    SEED { y 1.2 size 0.99 rotate 2.5 brightness 0.0015 }
}

rule SEED 0.04 {
    SQUARE {}
    SEED { y 1.2 s 0.9 r 1.5 flip 90 }
    SEED { y 1.2 x 1.2 s 0.8 r -60 }
    SEED { y 1.2 x -1.2 s 0.6 r 60 flip 90 }
}
```

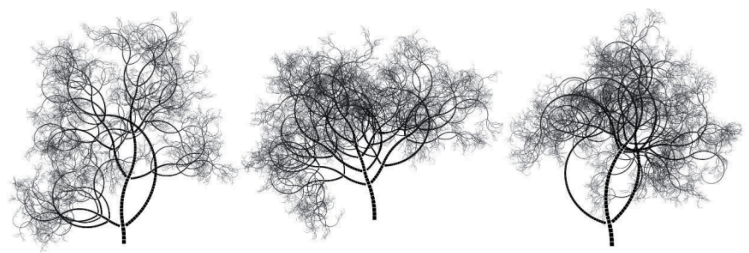


Figure 7: An example of a context-free Art system. The three structures are instances of the same system, but with different random seeds [6].

2.5.3 Shape Grammar

Another type of grammar where the geometrical interpretation is embedded intrinsically are Shape grammars [15]. Shape grammars are specifically designed for generating and manipulating shapes or spatial configurations. Unlike string-based grammars, such as L-systems and context-free Design grammars, which typically deal with symbols and strings, Shape grammars work directly with geometric entities. The rules in shape grammars are interpreted as operations on geometric shapes, and the resulting structures are explicitly geometric in nature.

A Shape grammar consists of rules that define the transformation or manipulation of shapes, such as translation, rotation, reflection and scaling. When applying the rules of a Shape grammar, the shapes are transformed or combined to create new shapes or configurations, and the process can be repeated iteratively to generate increasingly complex designs. Figure 8 illustrates a simple

Shape grammar.

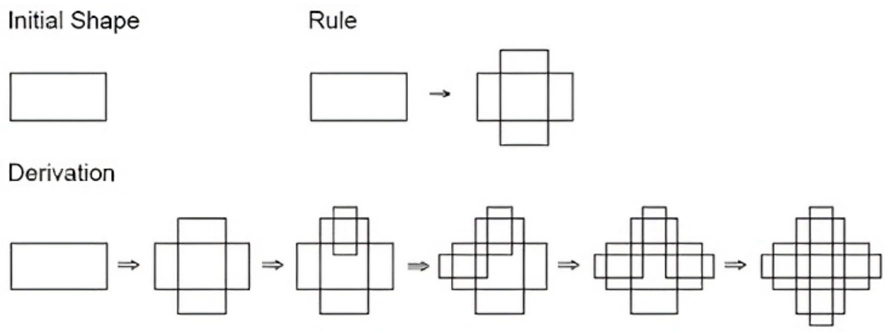


Figure 8: An example of Shape Grammar system [5].

2.6 Iterated Function Systems

Fractals [27] are complex geometric shapes that exhibit self-similarity and scale-invariance, meaning they have similar patterns or structures at different scales or magnifications. They are often generated using recursive processes and can have intricate, detailed, and infinitely repeating patterns. Fractals can be found in nature, such as in the patterns of ferns, coastlines, and snowflakes, as well as in various mathematical constructs. In generative art many different attempts have been made to generate fractals, as fractals often exhibit beautiful patterns. One of the more common methods to generate fractals are Iterated Function Systems (IFS) [4].

An IFS consists of a collection of contraction mappings, which are functions that transform points or shapes in a way that preserves their relative positions and distances but reduces their size. The most common technique to generate fractals using IFS is the random iteration algorithm also referred as ”Chaos Game” [4, 30], and it works as follows:

1. Start with an initial point or shape.
2. Randomly select one of the mappings from the IFS.
3. Apply the selected mapping to the point or shape.
4. Repeat steps 2 and 3 until a max number of iterations is reached.

The resulting points or shapes form a fractal pattern, characterized by self-similarity at different scales. Some famous fractals generated using IFS include the Sierpinski Triangle, Barnsley Fern (Figure 9), and Koch Snowflake (Figure 10).

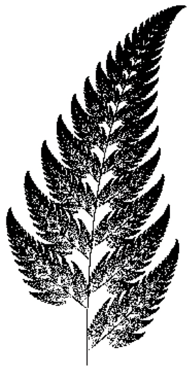


Figure 9: Barnsley Fern constructed by an IFS [45].

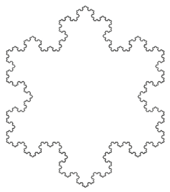


Figure 10: Koch Snowflake constructed by an IFS [36].

IFS and grammar-based methods have some similarities as that they both involve iterative processes and transformation rules to generate complex structures or patterns. The difference between them is that grammar-based methods, such as L-systems, Shape Grammars and context-free Design grammars, are rooted in formal language theory and involve the use of production rules to generate sequences or structures. IFS on the other hand, is a framework for generating fractals and self-similar structures using a set of contraction mappings. IFS are based on repeatedly applying these mappings to an initial point or shape, resulting in a complex structure with self-similar properties.

Other fractal generation methods also exist. For example the Mandelbrot set [27] and Julia set [8] are escape-time fractals, which are generated by iterating a complex function and determining whether points escape to infinity or remain bounded.

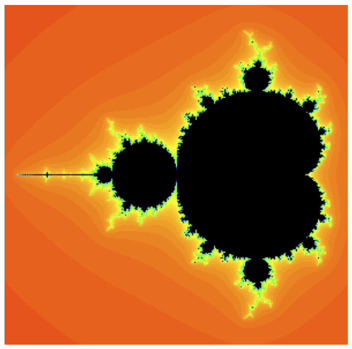


Figure 11: The Mandelbrot set [8].

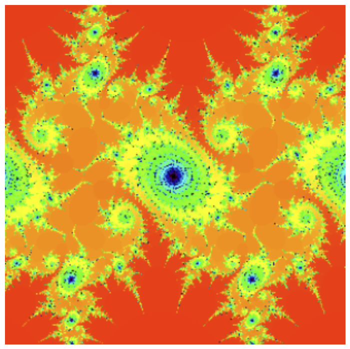


Figure 12: Julia set [8].

2.7 Cellular automata

Similar to grammar-based methods and Iterated Function Systems, cellular automata [18] are also rule-based systems and have been commonly used for generative art. Cellular automata are discrete models that simulate the behavior of a grid of cells evolving over time based on a set of rules. These rules depend on the states of neighboring cells and define how the cells change from one time step to the next. Cellular automata are often used to create animations that display the evolution of patterns over time. By defining a set of rules for the cells to follow, the artist can create animations that show the growth and development of patterns, from simple and orderly arrangements to more complex and chaotic forms.

While cellular automata share some similarities with grammar-based methods and IFS, such as the use of rules to generate patterns or structures, they differ in their fundamental structure and output. Grammar-based systems, such as L-systems or Shape grammars, often produce hierarchical or tree-like structures that resemble plants, branching patterns, or geometric shapes. IFS generate fractal images that exhibit self-similarity at different scales. Cellular automata can create a wide variety of patterns and behaviors, from simple repeating patterns to complex, dynamic systems that exhibit life-like behavior, like Conway's Game of Life [1].

Conway's Game of Life:

1. Any live cell with two or three live neighbours survives.
2. Any dead cell with three live neighbours becomes a live cell.
3. All other live cells die in the next generation.

Grammar-based systems can be context-free or context-sensitive, meaning the rules can depend on the surrounding context or the neighboring symbols/shapes. In IFS, the transformations are context-free, as they are applied independently of other transformations. Cellular automata rules are typically context-sensitive, depending on the immediate neighborhood of a cell.

2.8 Coupled equation systems

A coupled equation system [37] is a set of equations that describe the relationships between multiple variables. These equations can be either linear or nonlinear, and they may involve partial derivatives, integrals, and other mathematical operations. In a coupled equation system, the variables are interdependent, meaning that the value of one variable affects the values of the other variables in the system.

The primary difference between parameterized curves discussed in section 2.2 and coupled equation systems is that parameterized curves represent curves or paths in a coordinate system using a single parameter to trace the curve, while coupled equation systems model multiple interrelated variables and require simultaneous solutions to find the values of the variables involved.

Coupled equation systems are used to model a wide range of real-world phenomena, including fluid dynamics [43], electrical circuits [2], and biological systems [10]. In generative art, coupled equation systems have been used as a tool for creating dynamic and evolving visual forms [12]. By specifying a set of interdependent equations, artists can generate dynamic and complex shapes and patterns, that change and evolve over time.

2.8.1 Hénon map

The Hénon map [16, 42] is a discrete-time dynamical system. It is used to study chaos and the properties of chaotic systems and it is defined by the following coupled equations:

$$\begin{cases} x_{n+1} = 1 - a * x_n^2 + y_n \\ y_{n+1} = b * x_n \end{cases}$$

where:

- x_n and y_n are the coordinates of the point at iteration n.
- a and b are constants values.

When $a = 1.4$ and $b = 0.3$ the Hénon attractor is formed (see Figure 13). An attractor is a set of points in which a dynamical system tends to converge towards. The Hénon attractor is a strange attractor, which is a special type of attractor that has a fractal structure. Strange attractors are non-periodic, which means that the system's trajectory never exactly repeats itself. Strange attractors have been used in generative art due to their visually complex and captivating patterns.

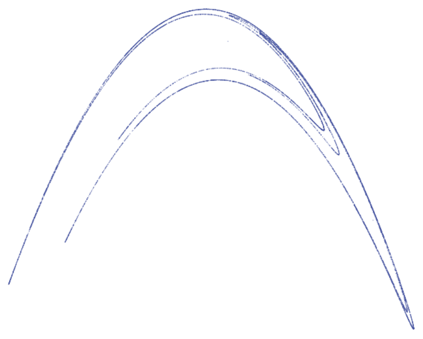


Figure 13: Visualization of the Hénon attractor [42].

2.8.2 Lorenz system

The Lorenz system [25, 11] is a set of three coupled differential equations that describe the dynamics of a simplified model of fluid convection in the atmosphere. The Lorenz system is is represented by the following equations:

$$\begin{cases} \frac{dx}{dt} = \sigma(y - x) \\ \frac{dy}{dt} = -xz + rx - y \\ \frac{dz}{dt} = xy - bz \end{cases}$$

where:

- x, y and z are the state variables.
- σ, r and b are constants values.

When $\sigma = 10$, $r = 28$ and $b = \frac{8}{3}$ the Lorenz attractor is formed (see Figure 14). The Lorenz attractor is classified as a strange attractor.

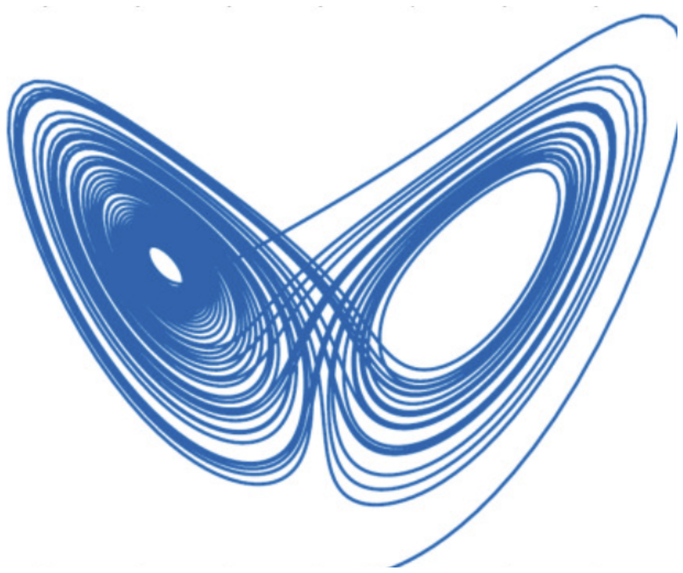


Figure 14: Visualization of the Lorenz attractor [11].

2.9 Agent Based Methods

Agent-based modeling [7] involves simulating the behavior of individual agents and their interactions to create complex patterns or behaviors, such as flocking, swarming [3], or crowd movement. In generative art, agent-based modeling can be used to create organic, emergent patterns and animations that result from the collective behavior of multiple agents.

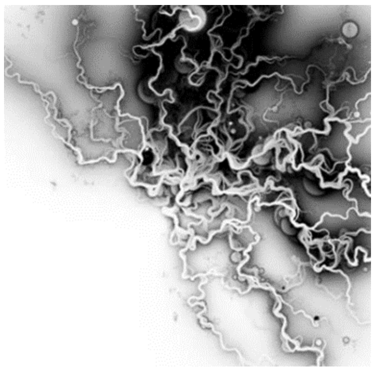


Figure 15: Magnetic Ink by Robert Hodgin (2007). This work was created using a flocking algorithm, where each agent leaves a mist of ink.

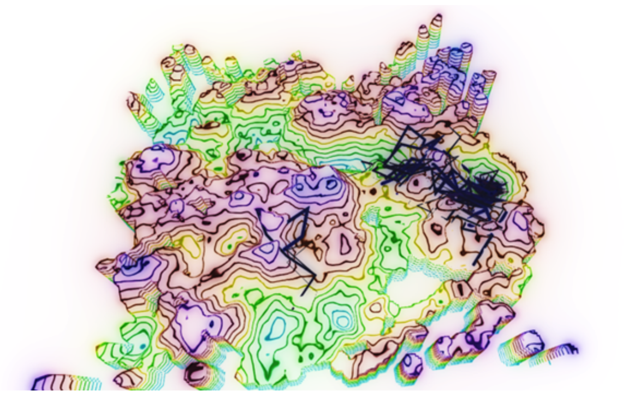


Figure 16: Generative art produced by Swarm Landscapes [3].

2.10 Particle systems

Particle systems simulate the behavior of particles to create natural phenomena like smoke, fire and flocking birds [35]. Each particle in the system has attributes like position, velocity, and lifespan, which can be updated according to a set of rules or forces. Particle systems are used in games to create dynamic, organic patterns and animations.

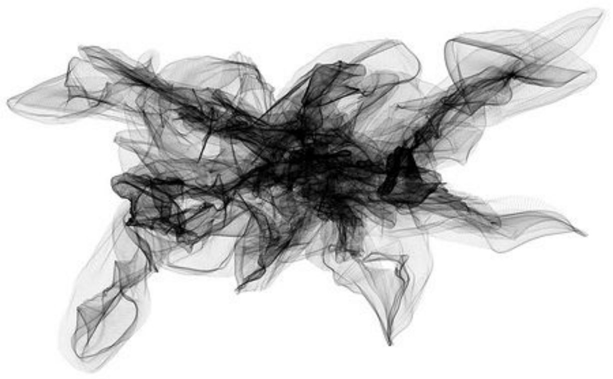


Figure 17: Drawing With Particles by 01010101 (2009). Jerome Saint-Clair, aka 01010101, fashioned this image with moves of the mouse, but the tool his mouse controlled was a generative particle system of 01010101's own creation.

2.11 Artificial neural networks

AI-generated art has gained significant popularity in recent years. AI-generated art refers to artwork generated with the assistance of artificial neural networks [14]. These neural networks can generate images, patterns, and other visual elements autonomously, based on learned patterns or given input data. An example of an AI-generated art application is DALL-E [34], which is an AI model developed by OpenAI that generates images from textual descriptions. Given a text prompt, such as "an armchair in the shape of an avocado" DALL-E can generate a wide range of images matching the description.

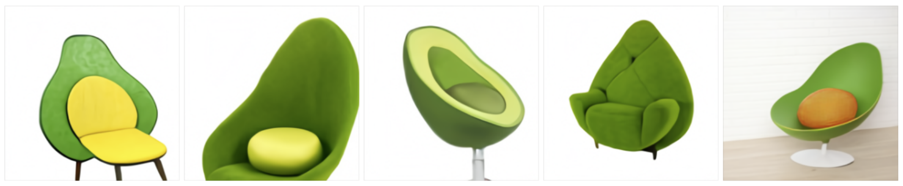


Figure 18: AI-generated art using DALL-E.

3 2D-Spiroplots

In 2020, Casper van Dommelen, Marc van Kreveld and Jérôme Urhausen introduced a new procedural dynamic system capable of creating intricate figures. The system functions by simultaneously rotating and plotting two points in an iterative process. They termed both the system and the resultant plot a "Spiroplot", which we refer to in this paper also as "2D-Spiroplot", because the generated plot exists in a two-dimensional space.

A 2D-Spiroplot is defined by a triplet (V, R, k) where V is a set of points defining the initial positions, R is a sequence of triplets (referred to as r-triplets) made up of two points from V and a given angle, and k is an integer representing the number of rotations. For each rotation, the current r-triplet's two points rotate around their midpoint by the defined angle.

As an example: Consider we have three points p_1, p_2 and p_3 with their respective initial coordinates (200, 400), (400, 400) and (400, 200). Suppose two r-triplets are given as $r_1 =(p_1, p_2,$ 90) and $r_2 = (p_2, p_3,$ 45). In the first iteration of the Spiroplot, the points p_1 and p_2 rotate counterclockwise by 90 degrees around their midpoint, with their new positions plotted on the canvas. In the next iteration the second r-triplet is used to rotate, in which the points p_2 (now at its updated position) and p_3 rotate counterclockwise by 45 degrees around their midpoint, with their new positions also plotted on the canvas. The iterations alternate between the two r-triplets until a total of k rotations are completed. With this simple iterative process beautiful figures can be surprisingly created.

Figure 19 demonstrates the outcome after 100,000 iterations, in which the plot color of p_1 is set to red, p_2 to green and p_3 to blue.

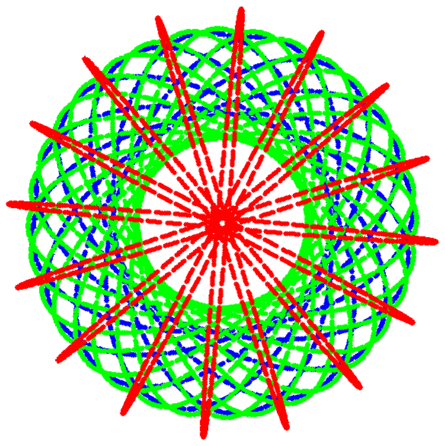


Figure 19: Spiroplot configuration with three points and two alternating r-triplets rotations of 90 and 45 degrees, after 100,000 rotations.

In the original paper on Spiroplots[9], the authors established the following inherent properties of 2D-Spiroplots:

- A rotation in a Spiroplot preserves the center of mass of the active points
- A rotation preserves the sum of squared distances to the center of mass of the active points.
- A rotation preserves the sum of squared distances between all pairs of active points.

These properties become pivotal in the 3D-Spiroplot section, where we experiment with various methods to generate 3D-Spiroplots, while trying to preserve these defining characteristics of a Spiroplot.

Additionally, the original paper demonstrated that the number of bits required to represent the points of a Spiroplot where r-triplets use rotation angles of 90 degrees and -90 degrees only, grows linearly in the number of rotations in the worst case. This may cause the effect that different Spiroplots will arise, when we use different numbers of bits to store and operate on numbers. In this study we will analyze the behaviour of Spiroplots under varying bit precision levels.

We also aim to determine the optimal bit resolution to produce the most accurate representation of a Spiroplot for the least cost. Furthermore, we have introduced new methods to visualize Spiroplots on screen.

3.1 Bit analysis of Spiroplots

Computers represent numbers in a binary (base-2) format and are constrained by finite memory. These factors inherently limit their numerical representation capabilities. As a result, when performing successive operations on numbers, precision of those numbers may reduce over time. Given these considerations and limitations, it raises the question if different Spiroplots will emerge when we use different datatypes with varying bit lengths to store and operate on numbers. From the original paper and user observation, it is evident that the Spiroplot is a chaotic type of system, where small changes in the input may result in completely different plots. To obtain a dense and aesthetic Spiroplot, typically thousands or even millions of rotations are needed to be performed. These rotations involve arithmetic and trigonometric operations on datatypes that store numbers with finite amount of bits, as a consequence rounding and finite precision errors may get accumulated over time. Before we analyze and experiment if using different datatypes, to store numbers with a different number of bits, would lead to different plots, we first dive deeper into the steps of performing a single iteration of a Spiroplot to identify potential points of rounding and finite precision error accumulation.

A Spiroplot is generated by iteratively rotating and plotting two points around their midpoint at the same time. For 2D point rotations, the 2D-rotation matrix is employed:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

This matrix represents a linear transformation that rotates points around the origin (0,0). Given an angle θ, the 2D-rotation matrix $R(\theta)$ rotates a point in the counter-clockwise direction around the origin with the angle θ. For its application to Spiroplots, it is essential to reposition the two points of the current r-triplet such that their midpoint coincides with the origin. This is achieved by offsetting each point by the coordinates of the midpoint. Upon utilizing the 2D-rotation matrix for the designated angle, the points are subsequently repositioned by adding back the original midpoint coordinates. This procedure ensures that the rotation is performed around the two points' midpoint instead of the origin. The full step of performing a single iteration of a Spiroplot given an r-triplet $R = (p_1, p_2, \theta)$, is as follow:

1. Calculate the midpoint m by averaging the coordinates of p_1 and p_2.
2. Offset p_1 and p_2 by subtracting m, aligning their new midpoint to the origin.

3. Rotate p_1 and p_2 using the 2D-rotation matrix with the angle θ.
4. Revert the translation by adding m back to the coordinates of p_1 and p_2.

In pseudocode:

Algorithm 1 Spiroplot midpoint rotation

```
**Input:** $p1 = (x1, y1)$, $p2 = (x2, y2)$, $\theta$
**Output:** Rotated coordinates of $p1$ and $p2$
$midpoint\_x \leftarrow (x1 + x2)/2$
$midpoint\_y \leftarrow (y1 + y2)/2$
$x1' \leftarrow x1 - midpoint\_x$, $y1' \leftarrow y1 - midpoint\_y$
$x2' \leftarrow x2 - midpoint\_x$, $y2' \leftarrow y2 - midpoint\_y$
$x1 \leftarrow \cos(\theta) \cdot x1' - \sin(\theta) \cdot y1'$
$y1 \leftarrow \sin(\theta) \cdot x1' + \cos(\theta) \cdot y1'$
$x2 \leftarrow \cos(\theta) \cdot x2' - \sin(\theta) \cdot y2'$
$y2 \leftarrow \sin(\theta) \cdot x2' + \cos(\theta) \cdot y2'$
$x1 \leftarrow x1 + midpoint\_x$, $y1 \leftarrow y1 + midpoint\_y$
$x2 \leftarrow x2 + midpoint\_x$, $y2 \leftarrow y2 + midpoint\_y$
```

In total 8 additions, 6 subtractions, 8 multiplications, 2 divisions, 4 cosine and 4 sine operations are performed for a single iteration of a Spiroplot. With a little optimization we can notice that all cosine operations yield the same value, and thus we only have to calculate $\cos(\theta)$ once and store the result, reducing the total cosine operations from 4 to 1. Similarly, the sine operations can be decreased from 4 to 1, with 2 additional sign bit flip operations to obtain the additive inverse. Reducing the total number of operations to perform a single rotation improves the performance of the system, but it also might help to reduce the strength of the growth of the accumulated precision errors when performing successive rotations.

We can reduce the total number of operations further by noticing that when the same r-triplet gets rotated in later iterations, the values of its 2D-rotation matrix remains unchanged unless the angle θ has been altered by the user during the iterations. This means that we can precalculate the rotation matrices of all r-triplets, which effectively eliminates the need for sine and cosine operations during rotations. These matrices only require updating when the r-triplet's angle is altered by the user. In total most often only 8 additions, 6 subtractions, 8 multiplications and 2 division operations are performed for a single iteration of a Spiroplot.

When considering the impact of rounding errors in floating-point arithmetic, some operations are more prone to introducing significant errors than others. Among the operations listed (addition, subtraction, multiplication, and division), subtraction and division are often the most problematic in terms of accumulating rounding errors, but for different reasons:

- Subtraction: The main issue with subtraction is catastrophic cancellation. This occurs when two nearly equal numbers are subtracted from one another, leading to a loss of significant digits.
- Division: Division can introduce rounding errors because the result of dividing two finite-precision numbers might not be exactly representable within the finite precision of the system. This can cause the result to be rounded to the nearest representable number, introducing an error.
- Multiplication: Multiplication can also introduce rounding errors, especially when multiplying numbers with large differences in magnitude. However, the nature of the error introduced by multiplication is often less problematic than that introduced by subtraction or division.
- Addition: In general, addition introduces the least amount of error among these operations, unless adding numbers with vastly different magnitudes, where the smaller magnitude number may not significantly impact the result due to the finite precision.

When only performing a small number of iterations it is reasonable to expect that Spiroplots of varying precisions would produce identical plots, given that that accumulated rounding errors are too small to have an impact of the final plotted pixel values. However, Spiroplots exhibit chaotic behavior, where minor input variations can lead to vastly different outputs. We conducted several experiments to determine whether minor rotation errors, accumulating over time, could similarly yield divergent plot results.

For the experiments we created five different datatypes to store numbers, each with a different number of bits. The different types of precision levels we analyzed are the following:

- Single precision (32-bit): about 7 decimal digits of precision.
- Double precision (64-bit): about 16 decimal digits of precision.
- Extended precision (80-bit): about 20 decimal digits of precision.
- Quadruple precision (128-bit): about 34 decimal digits of precision.
- Octuple precision (256-bit): about 71 decimal digits of precision.

Further details of the precision formats can be found in Table 1. We selected these five precision levels for the experiments based on the IEEE 754 standard, offering a broad bit-range to examine rounding errors across various bit depths.

We developed a C++ application to conduct these experiments, utilizing standard C++ data types like float and double for single and double precision calculations. For higher levels of precision we used the "MPFR" library to store and perform operations on the higher precision datatypes. The MPFR (Multiple

Precision	Sign bit	Exponent	Fraction	Total bits
Single	1 bit	8 bits	23 bits	32 bits
Double	1 bit	11 bits	52 bits	64 bits
Extended	1 bit	15 bits	64 bits	80 bits
Quadruple	1 bit	15 bits	112 bits	128 bits
Octuple	1 bit	19 bits	236 bits	256 bits

Table 1: Precision details

Precision Floating-Point Reliable) library provides a high-level way to perform arithmetic on floating-point numbers with virtually arbitrary precision. MPFR is built on top of the GMP (GNU Multiple Precision) library, which provides arbitrary precision arithmetic for integers and rationals. In our application, the angle of an r-triplet is stored in degrees, with the chosen level of precision. To utilize the 2D-rotation matrix, which requires angles in radians, we perform a conversion from degrees to radians while maintaining the same level of precision. The level of precision is also maintained in all subsequent arithmetic and trigonometric operations involved in the rotation.

To analyze whether varying input values influence the results, we have set up multiple Spiroplot configurations to examine if certain input combinations have an impact on the growth of the errors. We analyze the plots based on the following three categories: visual quality analysis, quantitative error analysis, and computational efficiency.

3.1.1 Visual quality analysis

To project a Spiroplot to the screen, the plot that it generates are stored in a grid. In the experiments we have set the cell size of the grid equal to the pixel size of the screen. This also means that when points are getting plotted, they have to convert their floating-point pixel coordinates to screen coordinates, by converting them to an integer. Rounding to the nearest integer method is applied in this conversion process.

In the visual quality analysis, we conducted a comparative analysis of the various precision plots to quickly identify any differences. We used 10 distinct Spiroplot configurations for this experiment, the same configurations are used for the other analytical methods as well. For each configuration, we generated five Spiroplots, each with the different level of precision. These plots were saved as integer grids and later exported as JPEG images for visual quality assessment.

The first configuration we tested on is what we define as the "standard" configuration. This configuration consist of three points p_1, p_2 and p_3, and two r-triplets $R_1 = (p_1, p_2, 90)$ and $R_2 = (p_2, p_3, 90)$. A total of three experiments were conducted using this configuration, with the iteration parameter k set to 100,000, 1 million and 10 million iterations, the results can be found in Table 2. From the results we immediately observe that the Spiroplot that uses the lowest precision (single precision), becomes gradually more different than the Spiroplots produced at higher precision. However no immediate noticeable difference can be seen between Spiroplots at the higher precision levels.

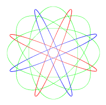
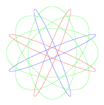
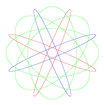
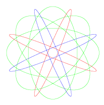
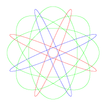
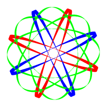
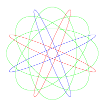
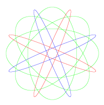
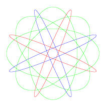
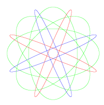
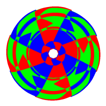
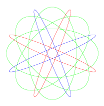
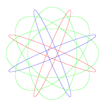
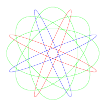
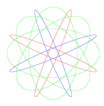
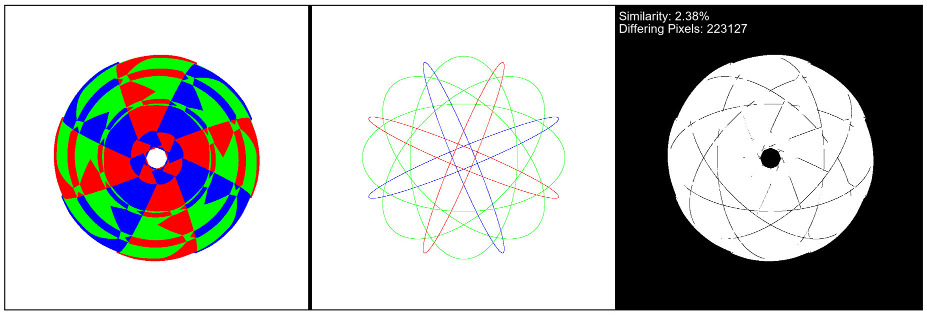
k	32-bit	64-bit	80-bit	128-bit	256-bit
100,000					
1,000,000					
10,000,000					

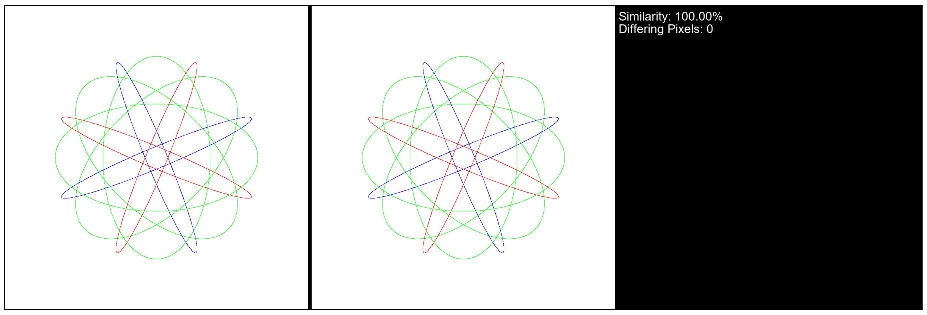
Table 2: The resultant plots of the standard Spiroplot configuration after 100,000, 1,000,000 and 10,000,000 iterations, produced by single, double, extended, quadruple and octuple precision.

To automate the process of identifying pixel-level differences across varying precision levels, we developed a Python script. This script compares two images, plots the disparities in pixel values onto a bitmap, and maintains a count of differing pixels. Each pixel location in the compared images is analyzed: if the pixel values diverge, the result is represented as a white pixel on the bitmap; otherwise, it appears as black.

Figure 20a provides an in-depth examination of pixel-level variances between the Spiroplot generated at the lowest precision level and the one produced at the highest precision level. It reveals that the number of differing pixels is 224,885 with a low similarity rate of 1.64%. Similarity is calculated by dividing the number of differing pixels with the total number of non-background pixels. The total count of non-background pixels is determined by the plot that covers the greatest number of unique pixels. Conversely, Figure 20b shows that when we compare the images of the Spiroplot generated at next lowest precision (double precision) with the one produced at the highest precision, no difference of pixel values can be found. This observation remains consistent when extending the comparison to Spiroplots generated with extended and quadruple precision lev-



(a) 32-bit vs 256-bit



(b) 64-bit vs 256-bit

Figure 20: Image comparison of the standard configuration after 10 million iteration.

els against those generated with octuple precision.

We further extended our tests to configurations where the rotation values were set very small, or where the initial points were situated very close or very far from one another. The objective was to ascertain if the initial input had an impact on the growth of the precision errors. However, across all tested Spiroplot configurations, a consistent trend emerged: only the Spiroplots generated at the lowest precision (single precision) displayed any significant differences when compared to those produced at higher precisions. This observation leads us to question whether we have set the number of iterations too small, to detect differences in the higher-precision Spiroplots, prompting further investigation into the required number of iterations for such differences to become noticeable. This aspect will be addressed in Section 3.1.2.

3.1.2 Quantitative error analysis

In the original paper, the authors proved that certain Spiroplots are cyclic. Cyclic Spiroplots possess the unique characteristic that, upon completing a specific number of iterations, the coordinates of all points revert to their initial positions. The cycle length of a Spiroplot is defined as the minimum positive integer representing the number of complete rotations such that all points return to their original coordinates. Knowing the configuration and cycle length of a cyclic Spiroplot enables the precise analysis of error by executing iterations in multiples of the cycle length. For the various bit-precision levels, deviations between generated points' positions and their original coordinates can be calculated using Euclidean distance as the error metric. However, we have chosen to forego the analysis of cyclic Spiroplots in this study as they tend to yield less intriguing plots compared to their non-cyclic counterparts. Additionally, finding configurations for cyclic Spiroplots proves challenging.

For non-cyclic Spiroplots, the exact final coordinate positions after numerous iterations remain unknown in advance. Hence, the error in positions will be measured against the point's positions of the Spiroplot generated at the highest precision level. In our analysis, each non-cyclic Spiroplot undergoes 10 million rotations, with coordinate positions saved to a CSV file at every 100-rotation interval. This 10-million-rotation limit was set due to constraints related to disk writing and storage capacity, as octuple-precision data occupied gigabytes of disk space. The saving interval was strategically chosen to balance disk resource conservation with the detailed tracking of error progression.

After generating the files that recorded the evolution of the coordinate positions over time, we analyzed this data using the R programming language. The "readr" and "Rmpfr" libraries were used for data importation. To be able to perform an analysis between various precision levels, we had to pad each precision level, except for the highest, with additional zeros to match it with the highest precision level.

To evaluate the error between various bit-precision levels, the following comparison methods have been employed:

- Compare the x and y coordinate error of each point in a Spiroplot separately.
- Compare the combined x and y coordinate error of each point in a Spiroplot.
- Compare the error of the center of mass.

We continue to use our standard configuration for this analysis, setting k to 10 million iterations.

Separate x and y error

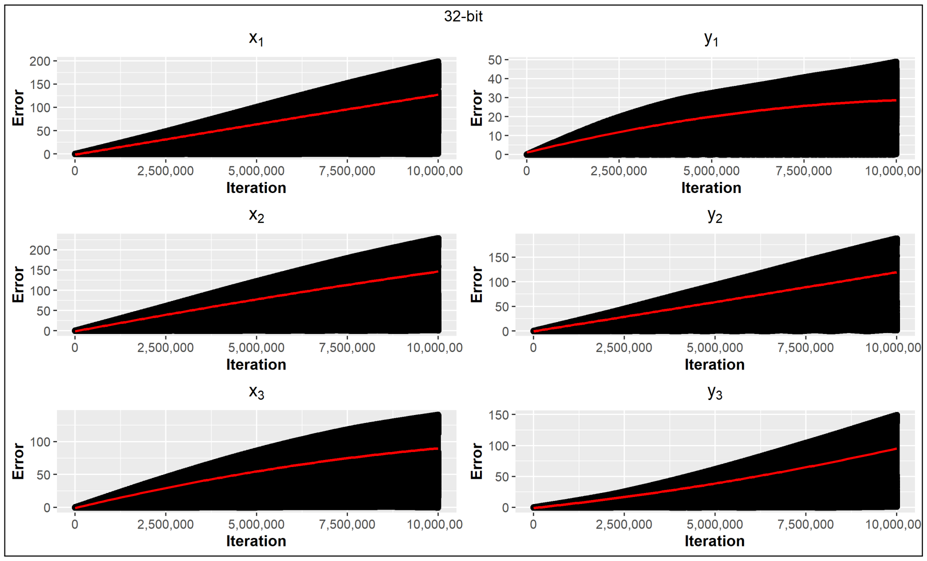
In this method we compare the x and y coordinates of each point to those of the same point at the highest precision level in each iteration. The absolute difference in the x and y positions is used as our error metric. These error values are plotted on a graph, with the number of iterations on the x-axis and the error values on the y-axis. A red quadratic curve is fitted to the data points to help interpret the trends. The results of this analysis can be found in Figure 21 and Figure 22. It's important to note that in our analysis, an error value of one is equivalent to the size of a single pixel.

In Figure 21a the absolute difference of the coordinates of single precision and octuple precision points is presented. We observe that the errors in single precision rapidly grows beyond a single pixel size, another observation we make is that not all errors grow at a uniform rate. The fitted quadratic curve suggests that the errors in the x and y-coordinates of the first and second point grows at a linear rate. Conversely, the growth rate of the x-coordinate error for the third point appears to diminish over time. This deceleration, however, is counterbalanced by an accelerating growth in its corresponding y-coordinate error.

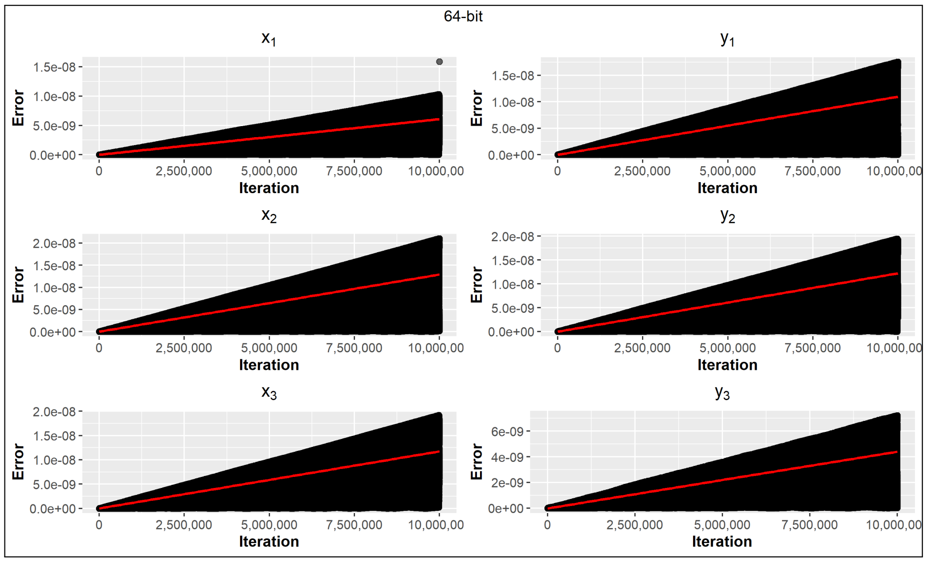
In figure 21b we find the results when we compare the x and y coordinates of double precision with octuple precision. In contrast to single precision, the graphs shows that the error values for both x and y-coordinate remain below a pixel size, even after 10 million iterations. Moreover, all errors seem to grow at a consistent linear rate. In Figure 22a and Figure 22b the results of this analysis on extended and quadruple precision can be found. We observe that the graphs that they produce are very similar to the ones produced by double precision, with the only noticeable difference being the error magnitudes. For instance, after 10 million iterations, most errors in double precision are roughly 10^{-8} pixel distances. In extended and quadruple precisions, the corresponding errors are around 10^{-16} and 10^{-31} pixel distances, respectively. Remarkably, beyond double-precision, the error growth appears to decrease by an order of magnitude that is nearly double that of the previous, less precise level. To elaborate, the magnitude of 10^{-8} is twice that of 10^{-16}, and the magnitude of 10^{-16} is nearly twice of 10^{-31}.

Across all levels of precision, the error growth pattern appears to resemble a random walk. The range of error broadens over time but stays confined within a triangular boundary. Moreover, even after 10 million iterations, we find instances where the error momentarily reverts close to zero. This behavior can be explained by cancellation of errors, wherein subsequent operations introduce errors that effectively negate prior errors. For example, if one operation introduces a small positive error and the next introduces a roughly equivalent negative error, the cumulative error might decrease.

Lastly, for each precision level, we used a linear predictive model to estimate when the absolute distance in the x or y-coordinates would deviate by one whole pixel from the coordinates calculated at the highest precision level. The results of these predictions are summarized in Table 3.

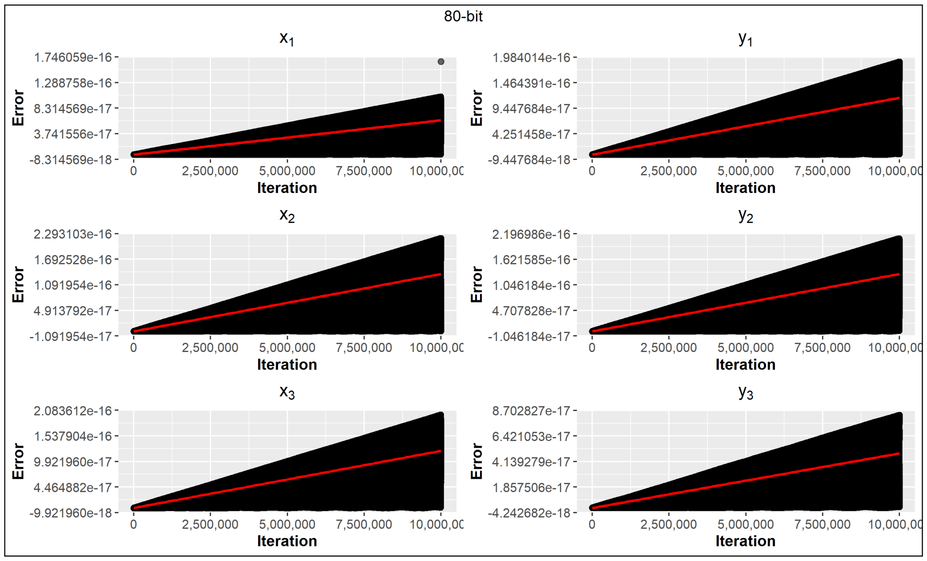


(a) Single precision

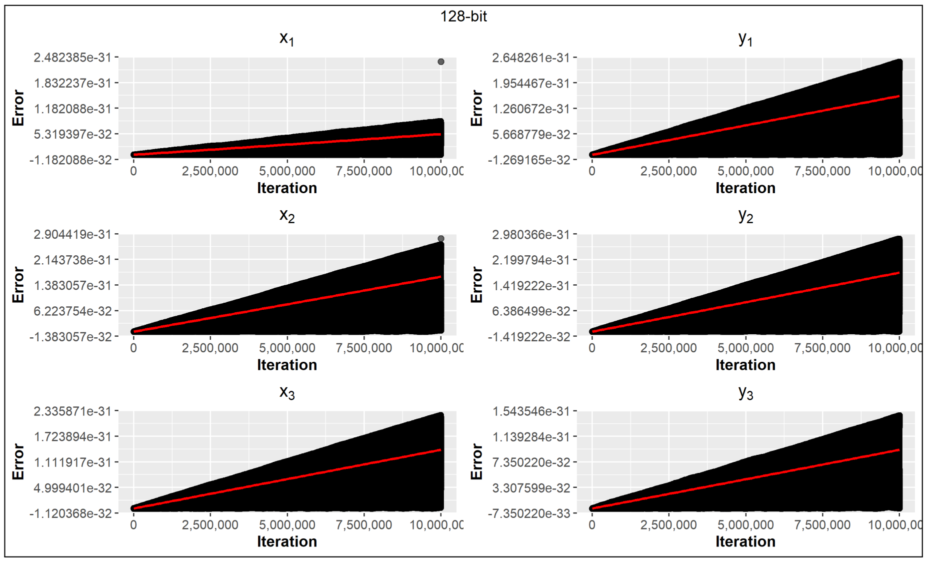


(b) Double precision

Figure 21: Plots of the absolute differences between the coordinates generated using single and double precision, each compared to octuple precision, using the standard Spiroplot configuration.



(a) Extended precision



(b) Quadruple precision

Figure 22: Plots of the absolute differences between the coordinates generated using extended and quadruple precision, each compared to octuple precision, using the standard Spiroplot configuration.

Table 3: Absolute difference predictions (in iterations) exceeding one.

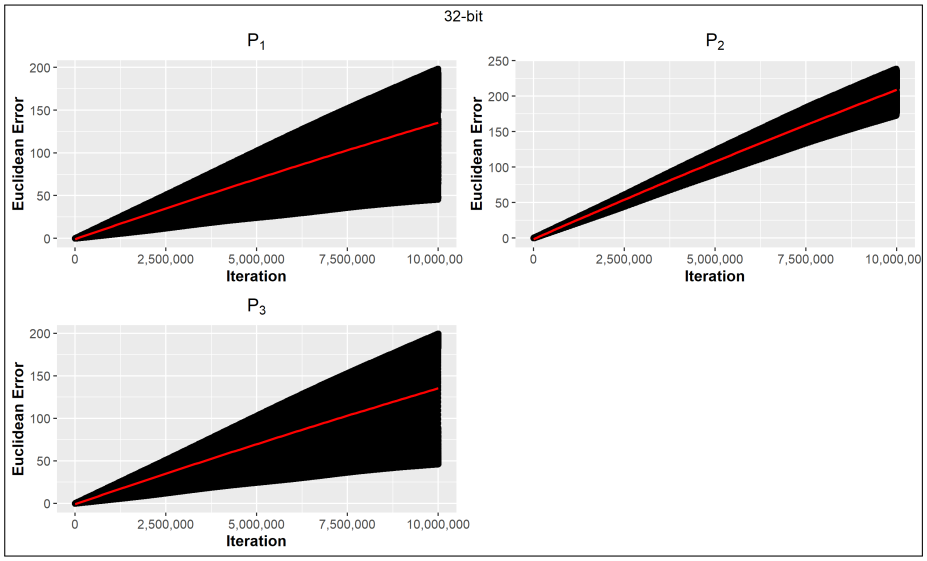
	32-bit	64-bit	80-bit	128-bit
X_1	128779.6	1.650515×10^{15}	1.630191×10^{23}	1.904573×10^{38}
Y_1	1289341	9.109434×10^{14}	8.628398×10^{22}	6.259067×10^{37}
X_2	74653.47	7.747830×10^{14}	7.438590×10^{22}	6.127627×10^{37}
Y_2	167252.4	8.176489×10^{14}	7.736661×10^{22}	5.546338×10^{37}
X_3	544315.2	8.520997×10^{14}	8.214304×10^{22}	7.127236×10^{37}
Y_3	785259.5	2.259711×10^{15}	2.047849×10^{23}	1.077526×10^{38}

Combined x and y error

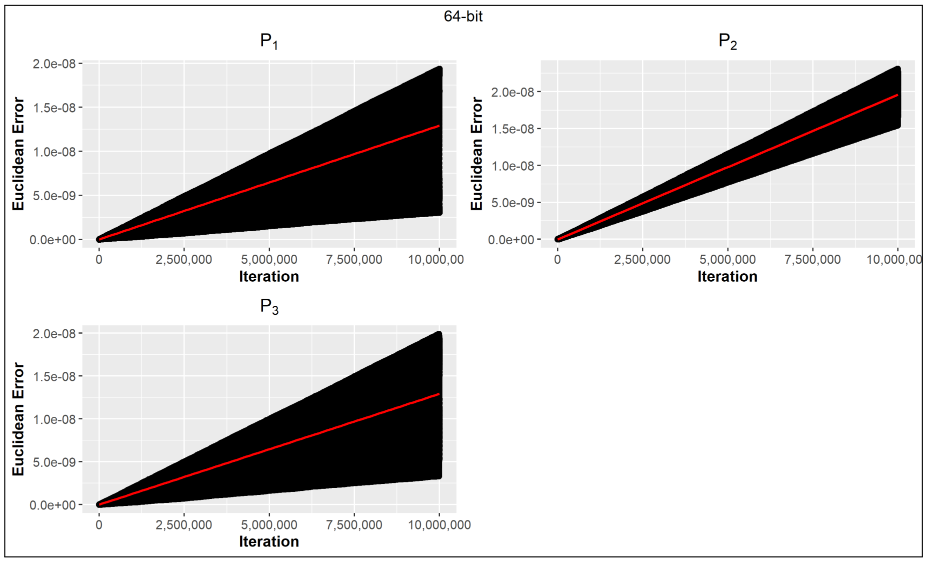
In the second analytical approach, we compute the Euclidean distance between points generated at various precision levels with their corresponding points at the highest precision level. The results are displayed in Figure 23 and Figure 24. We observe that all precision levels appears to produce similar graphs, with the only exception being the magnitude of the error. Similar to the previous method the average error appears to increase at a consistent rate, with the range of the errors being confined within a triangular boundary. The magnitude of the errors in each precision level also appears to be similar of that in the result of the previous analysis.

What sets this method apart is the observation that the minimum error also incrementally increases over time. This suggests that the phenomenon of ”error cancellation” has diminished influence when both x and y-coordinates are utilized to compute differences with higher-precision points. Consequently, a persistent level of Euclidean error begins to accumulate after a certain number of iterations. Given the linear growth rate of this minimum error, we can confidently assert that pixel-level differences will emerge after a predetermined number of iterations.

For all precision levels we have predicted, using a linear model, of when the Euclidean distance of a point would be a whole pixel distance away from the coordinates of the same point produced by the highest precision. This implies that beyond this threshold of iterations, we would anticipate slight differences in the integer grids of the Spiroplot compared to those generated at the highest precision level. Table 4 summarizes these predictive outcomes. For double precision, our model estimates that pixel-level difference would become noticeable around 10^{14} iterations when compared to the Spiroplot created using octuple precision. Given the astronomical scale of this iteration count, it is practically safe to conclude that double precision is more than adequate for generating a highly accurate Spiroplot.

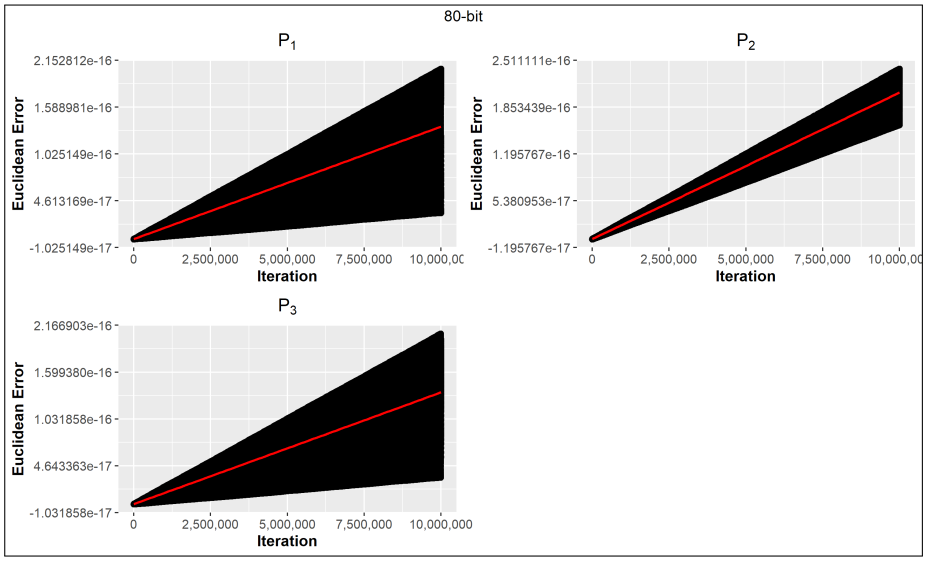


(a) Single precision

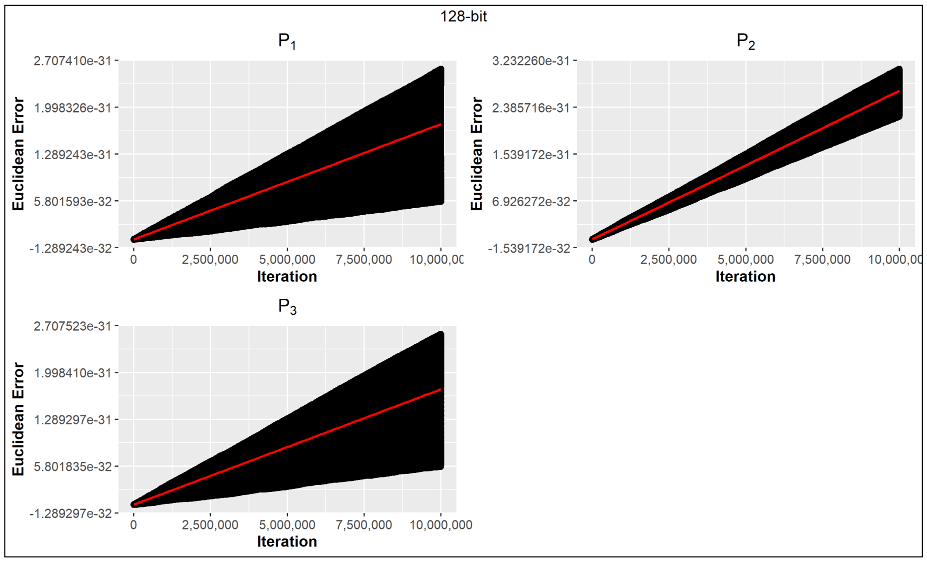


(b) Double precision

Figure 23: Plots of the Euclidean distance between the coordinates generated using single and double precision, each compared to octuple precision, using the standard Spiroplot configuration.



(a) Extended precision



(b) Quadruple precision

Figure 24: Plots of the Euclidean distance between the coordinates generated using extended and quadruple precision, each compared to octuple precision, using the standard Spiroplot configuration.

Table 4: Euclidean distance predictions (in iterations) exceeding one.

	32-bit	64-bit	80-bit	128-bit
P_1	17138.04	7.734128×10^{14}	7.38492×10^{22}	5.731832×10^{37}
P_2	6952.826	5.097464×10^{14}	4.859343×10^{22}	3.724126×10^{37}
P_3	16870.25	7.736415×10^{15}	7.384624×10^{22}	5.731146×10^{37}

Center of Mass Error

In the original paper it was proven that the center of mass of the active points is preserved after performing a rotation. Based on this evidence, it is justifiable to use deviations from the original center of mass as an error metric. During each iteration, we computed the Euclidean distance between the center of mass at a given precision level and the center of mass at the highest precision level, the result are shown in Figure 25. Fitting a red quadratic curve to these data points, we observed that the center's error increases over time. Notably, the error growth approached linearity as the precision level was ramped up, becoming unstable at a threshold of 128 bits. The underlying cause of this instability remains undetermined.

	32-bit	64-bit	80-bit	128-bit
Iterations	6,011,017	1.320892×10^{14}	9.80041×10^{22}	4.137295×10^{37}

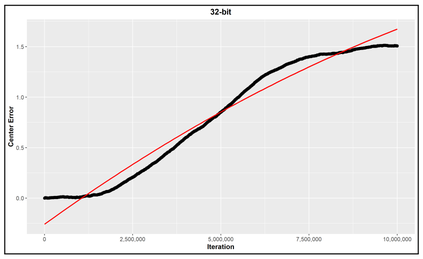
Table 5: Center error predictions exceeding one.

In conclusion, double precision is typically sufficient for generating accurate Spiroplots when fewer than 10^{14} iterations are performed. For more extensive runs, the Spiroplot grid tends to remain constant or appear to repeat, likely due to integer conversion issues. Higher precision levels become necessary only when setting extremely small rotation angles or significantly enlarging the grid, such that each cell represents just a fraction of a pixel.

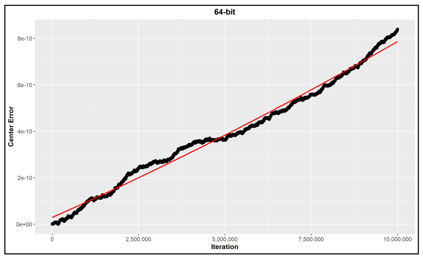
3.1.3 Computational Efficiency

We investigate the trade-off between bit precision and computational efficiency in the generation of Spiroplots. While higher bit precision can improve accuracy, it also tends to incur a greater computational cost, raising the question of optimal trade-offs.

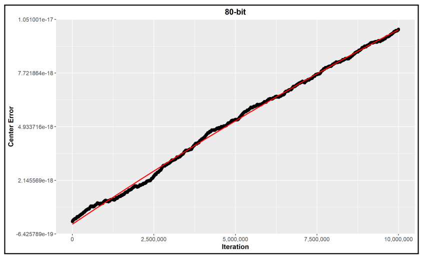
Our previous analysis indicates that double precision is generally sufficient for generating an accurate Spiroplot for up to 10^{14} iterations. For applications demanding higher certainty however, one may opt for even higher precision levels. For this scenario we measured the time required to generate Spiroplots at various precision levels. These tests were performed on the standard Spiroplot configuration, on an Intel® Core™ i7-7700K CPU, operating at 4.2 GHz, and 16



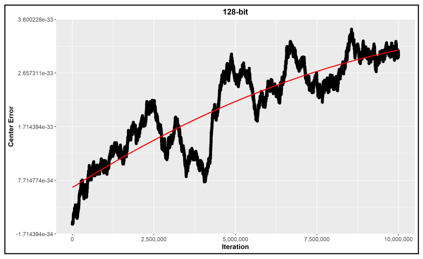
(a) Single precision



(b) Double precision



(c) Extended precision



(d) Quadruple precision

Figure 25: Graphs of the Euclidean distance between the center of mass of the generated points using various bit precision levels, each compared to octuple precision, using the standard Spiroplot configuration.

GB of DDR4 RAM running at 2400 MHz. The result can found in Figure 26, where we plotted the computational time against the total number of iterations for each precision level. From the graph we note two key observations:

1. Across all precision levels, the time complexity for generating a Spiroplot is linear with respect to the number of iterations. This implies that irrespective of the precision level, the computational time scales linearly as the number of iterations increases.

2. There is a distinct divergence in computational time between single and double precision with the higher precision levels. Single and double precision calculations are almost identical in terms of time required, whereas a significant increase is observed for extended, quadruple, and octuple precision levels.

The near-identical performance of single and double precision levels can be explained by the fact that we utilized standard C++ data types for single and double precision, benefiting from highly optimized arithmetic functions available in standard C++. On the other hand, the increase in computational time for higher precision levels is likely due to the overhead associated with using the "MPFR" library to store and operate on higher precision numbers, which is not as optimized for the specific arithmetic operations required for Spiroplot generation. Interestingly, extended, quadruple, and octuple precision levels also exhibit near-identical computational times, suggesting minimal downside in choosing the highest available precision when more than double precision is needed.

These and the previous results provide valuable insights into the trade-offs between computational time and precision, suggesting that unless higher precision levels are strictly necessary, double precision is more than sufficient given their superior computational efficiency and highly accurate results.

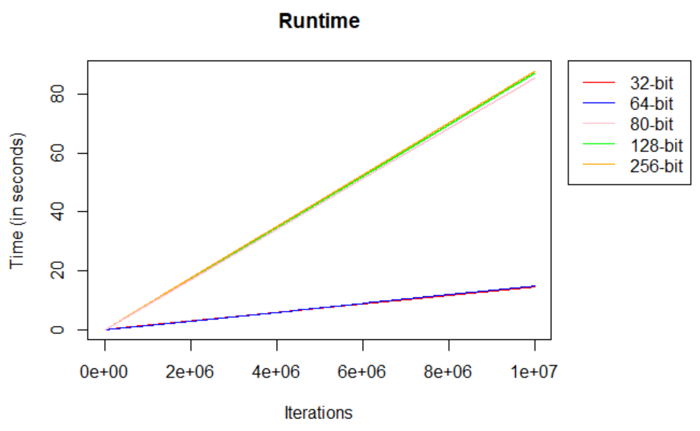
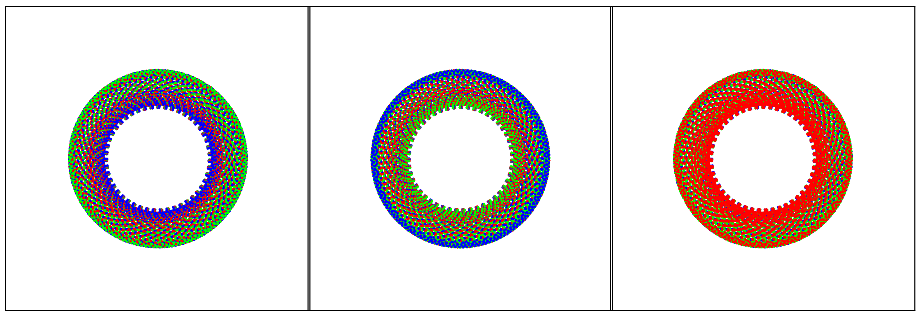


Figure 26: Graph of the runtime analysis of the various precision levels.

3.2 Enhancements to the original Spiroplot application

While developing the application, we introduced additional methods for determining point-drawing order. In scenarios where points overlap at identical pixel locations, deciding the sequence in which they are plotted becomes crucial. In the original application, each initial point had its dedicated grid to store its plotted locations. Grid overlaps were determined using a fixed point-list order, with the grid corresponding to the last point in the list overlaying all preceding ones.

We have introduced two new ordering methods: Last-In, First-Out (LIFO) and First-In, First-Out (FIFO). In the LIFO approach, newer points are drawn over existing ones. In the FIFO approach, once a pixel location is claimed by a point, it remains occupied, preventing other points from overwriting it. These methods offer greater variety in Spiroplot generation, as seen in Figure 27. A notable advantage is that they utilize a singular grid, leading to memory resource conservation. With the improved efficiency of the application, the LIFO method can generate real-time animated Spiroplots as illustrated in Figure 28.



(a) LIFO (b) FIFO (c) point-list order

Figure 27: Three generated Spiroplots using the same configuration with different point-drawing order methods.

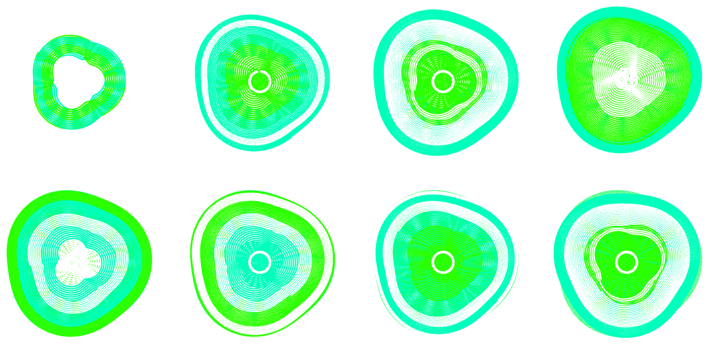


Figure 28: Animated sequence of a Spiroplot configuration with r-triplets $(p_1, p_2, 3)$, $(p_1, p_3, 3)$ and $(p_1, p_4, 3)$, using the LIFO point-drawing order method.

4 3D-Spiroplots

The second part of this paper delves into the extension of Spiroplot into three dimensions. The original 2D-Spiroplots are generated by iteratively rotating two points around their shared midpoint at the same time. This approach offers the attractive feature of maintaining the center of gravity of all active points, even after a rotation. It also preserves the sum of squared distances to the center of mass of the active points, as well as the squared distances between all pairs of active points. These properties lend a level of stability to the inherently chaotic system, resulting in a higher likelihood of generating dense aesthetic plots. As we venture into the realm of 3D-Spiroplots, we aim to retain these valuable characteristics. However, rotating two points around their midpoint in a three-dimensional space presents complications, notably the need to define a rotation axis. This paper proposes several techniques for executing rotations on pairs of points in a three-dimensional space. To test if any of these methods could generate beautiful 3D figures, we developed an application that can generate and visualize these 3D-Spiroplots.

4.1 Rotation methods

In our quest to design 3D-Spiroplots, we used an auxiliary point that remains consistent across every r-triplet to help define the rotation axis. We have selected the center of mass of all active points for this role. Similar to the original 2D-Spiroplot, each designed 3D-Spiroplot is defined by a triplet (V, R, k) where V is a set of 3D points defining the initial positions, R is a sequence of triplets made up of two points from V and a given angle, and k is an integer representing the number of rotations. Additionally, we choose a rotation method to determine the rotation axes for the two points in the current r-triplet. In each iteration, these points rotate around their predetermined directed axes by the specified angle, adhering to the right-hand rule.

We have designed four distinct methods for defining the rotation axes:

1. Vector from the center of mass to the midpoint as the directed axis
2. Axis parallel to one of the position vectors
3. Multiple position vectors axis rotation
4. Normal vector of a plane as the directed axis

For illustrative purposes, we consider a 3D-Spiroplot configuration where only the points of the current r-triplet $R = (P_1, P_2, \theta)$ are displayed, along with the center of mass of all active points C and the midpoint M of P_1 and P_2.

4.1.1 Vector from the center of mass to the midpoint as the directed axis

In this initial approach we took the normalized vector from the center of mass (of all active points) to the midpoint of P_1 and P_2 as the rotation axis. The resulting plots, as illustrated in Figure 30, reveal that each point generates new coordinates on spheres centered around the center of mass. Through experimentation, we found that the input rotation angles influence the filling patterns of these spheres, while the initial coordinates dictate their radius. When all the spheres are set to the same size it yields intricate patterns on a single spherical object, as seen in Figure 30. For instance, the bottom-right image showcases a pattern resembling a yarn ball.

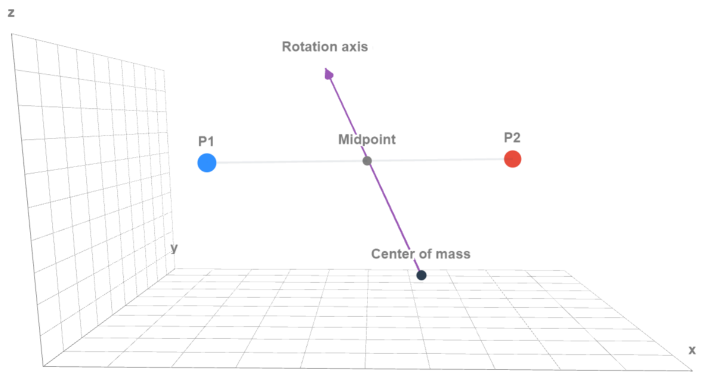


Figure 29: Rotation axis as an vector from the center of mass to the midpoint of P_1 and P_2.

4.1.2 Axis parallel to one of the position vectors

We can choose the rotation axis as an axis that goes through the midpoint of P_1 and P_2 and is parallel to one of the position vectors of P_1 and P_2. It should be noted that while the term ”position vector” traditionally refers to a vector originating from the coordinate system's origin to a point in space, in this paper we denote a position vector as an vector originating from the center of mass to a point in space. For instance, the normalized vector from the center of mass to P_1 could serve as the chosen position vector. Accordingly, the rotation axis would pass through the midpoint of P_1 and P_2 and remain parallel to this normalized vector. The outcomes of this approach can be found in Figure 33.

One notable difference with the previous method in the generation of the plots, lies in the behavior of the rotated points. Specifically, these points tend to converge over time, eventually aligning along a single line. To generate dense aesthetic plots using this method, very small rotation angles must be set. The decision to rotate around either the first or the second point of the r-triplet also influences the outcome of the resulting plots for the given configuration. Our experiments reveal that this method frequently produces ovoid objects with helical patterns.

4.1.3 Multiple position vectors axis rotation

Both position vectors can be used during the rotation process. P_1 can be rotated around the vector that goes through the midpoint and is parallel to the unrotated position vector of P_2, while P_2 rotates around the vector that goes through the midpoint and is parallel to the unrotated position vector of P_1. The results can be seen in Figure 35.

Similar to the preceding method, the active points tend to converge over time,

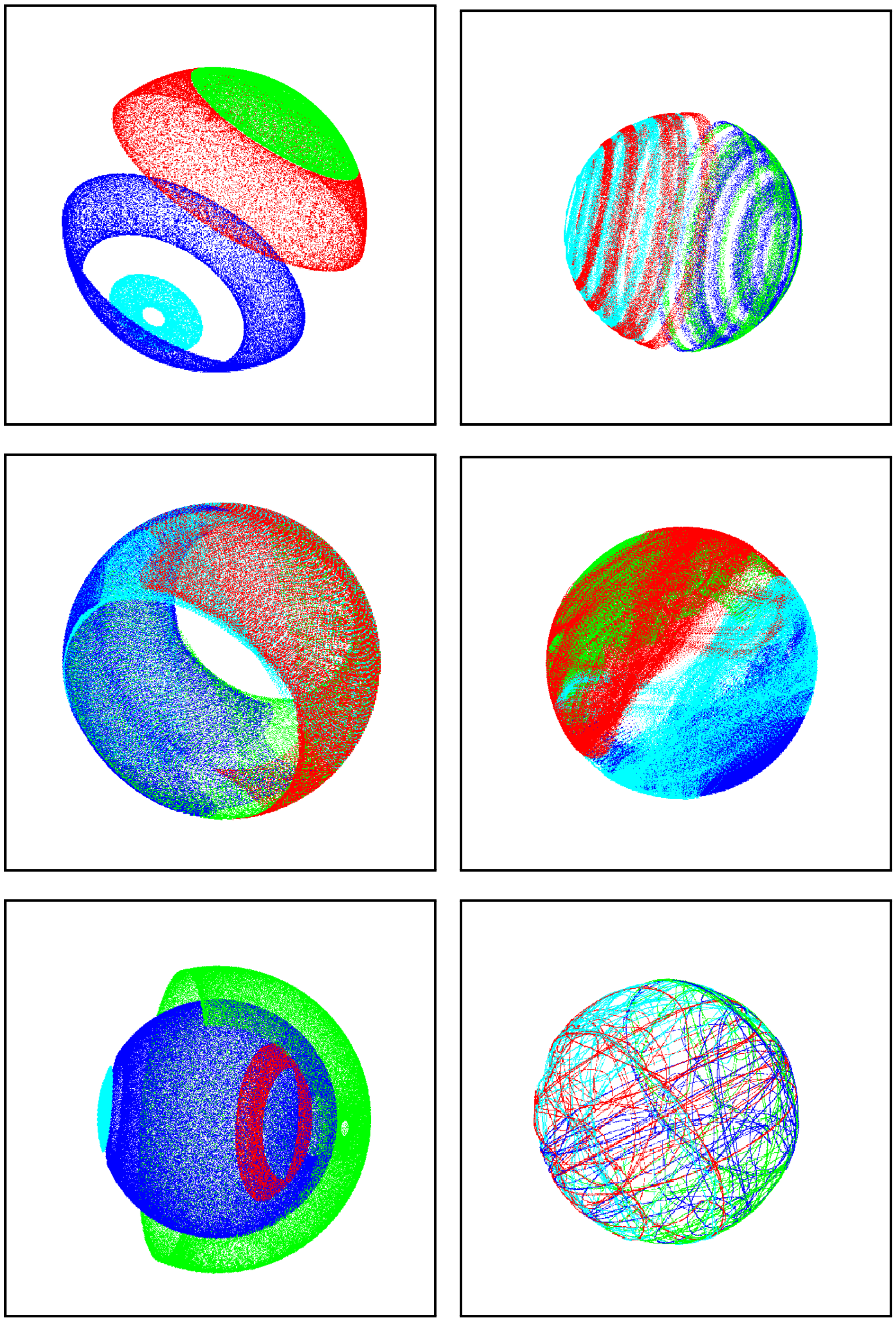


Figure 30: 3D-Spiroplots generated by the first rotation method.

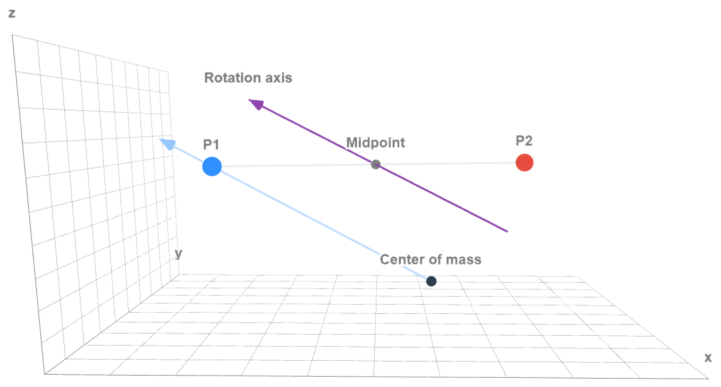


Figure 31: Rotation axis as an axis that goes through the midpoint of P_1 and P_2 and is parallel to the position vector of P_1.

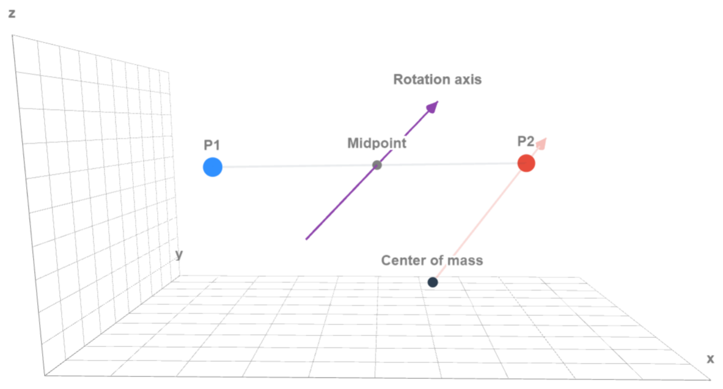


Figure 32: Rotation axis as an axis that goes through the midpoint of P_1 and P_2 and is parallel to the position vector of P_2.

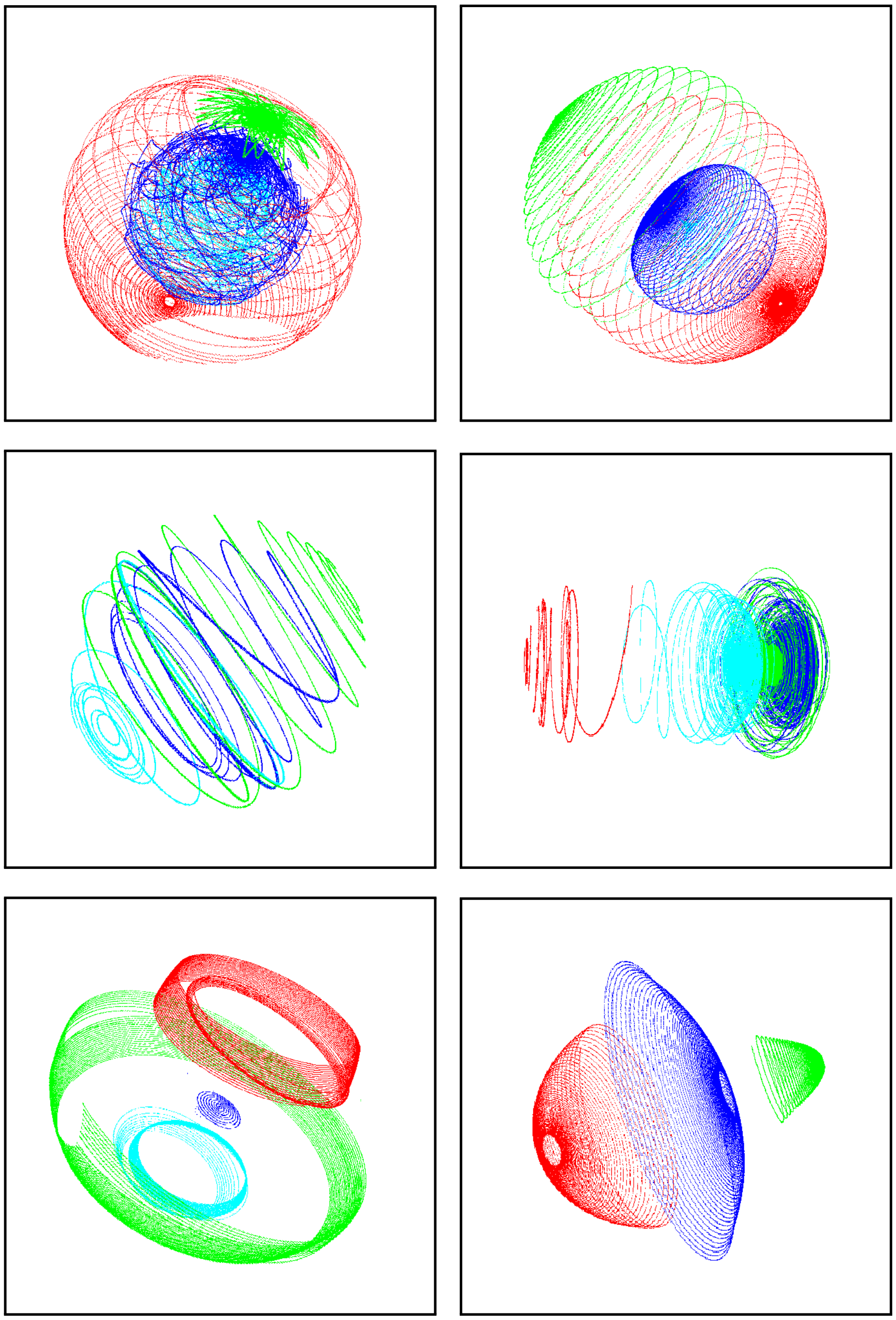


Figure 33: 3D-Spiroplots generated by the second rotation method.

ultimately aligning along a single line. This behavior eliminates the need to figure out a stopping point for the iteration count, as the convergence process occurs rapidly, thereby serving as an advantageous feature for this and the previous method. From our experiments, we conclude that this method yields the most intricate and diverse plots. Moreover, a significant portion of these generated plots exhibit symmetry, which inherently make the plots more visually appealing.

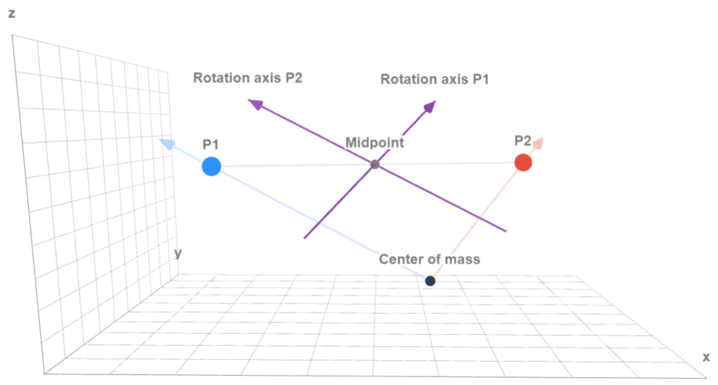


Figure 34: Rotation axes of P_1 and P_2, where P_1 gets rotated around the axis that goes through the midpoint of P_1 and P_2 and is parallel to the position vector of P_2, while P_2 gets rotated around the axis that goes through the midpoint of P_1 and P_2 and is parallel to the position vector of P_1.

4.1.4 Normal vector of a plane as the directed axis

A plane can be defined by the two rotating points P_1 and P_2, and the center of mass M. A normal vector to the plane can be defined by taking the cross product of the position vectors of P_1 and P_2. After normalizing the resulting vector, the normal vector (resulting from $P_1 \times P_2$) can then be used as the rotation axis for the two points.

Similar to the first method, each points generate new coordinates on the surface of their own respective sphere that are centered around the center of mass. Unlike the first method, this approach results in more evenly distributed points across the surface of the sphere as seen in Figure 37.

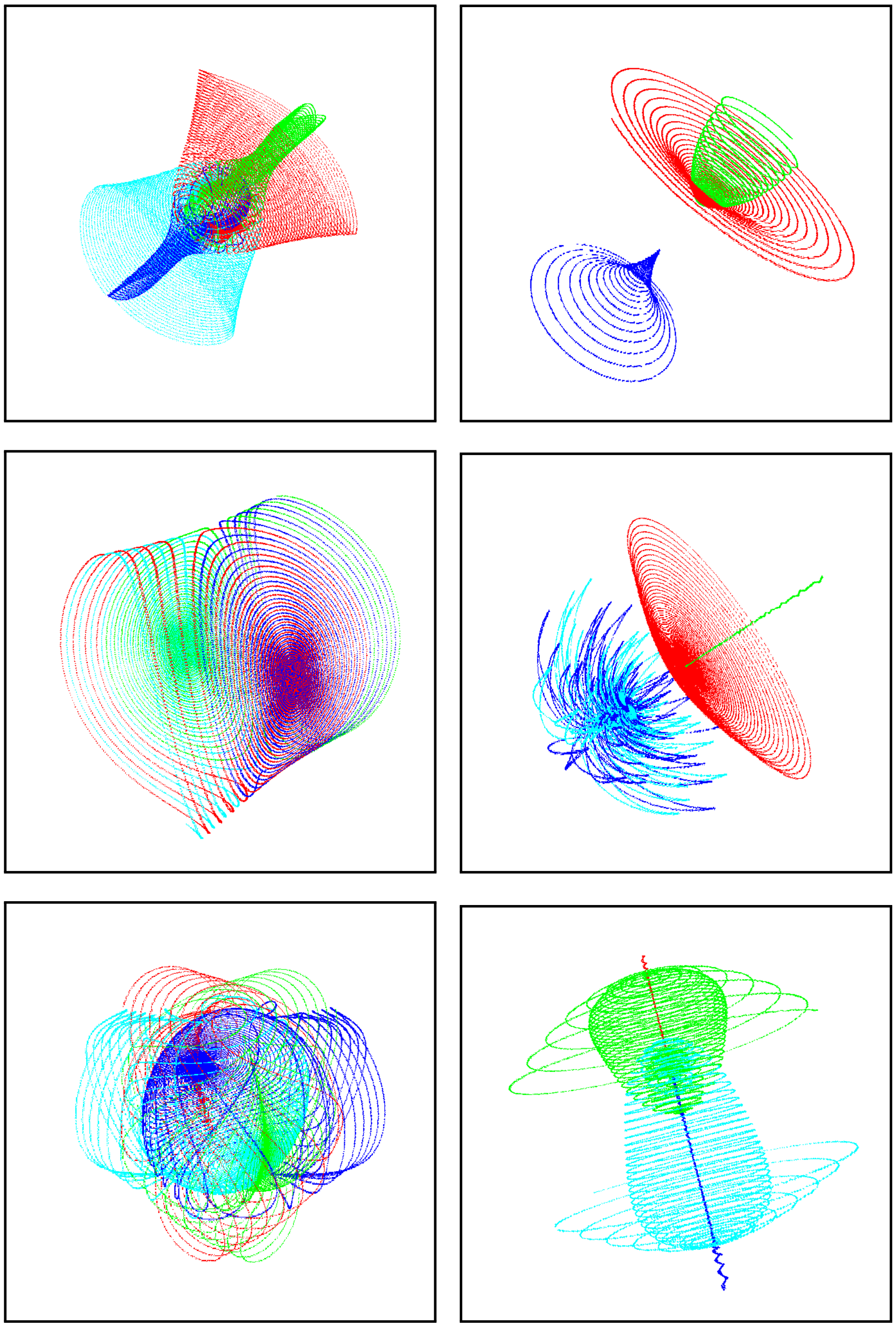


Figure 35: 3D-Spiroplots generated by the third rotation method.

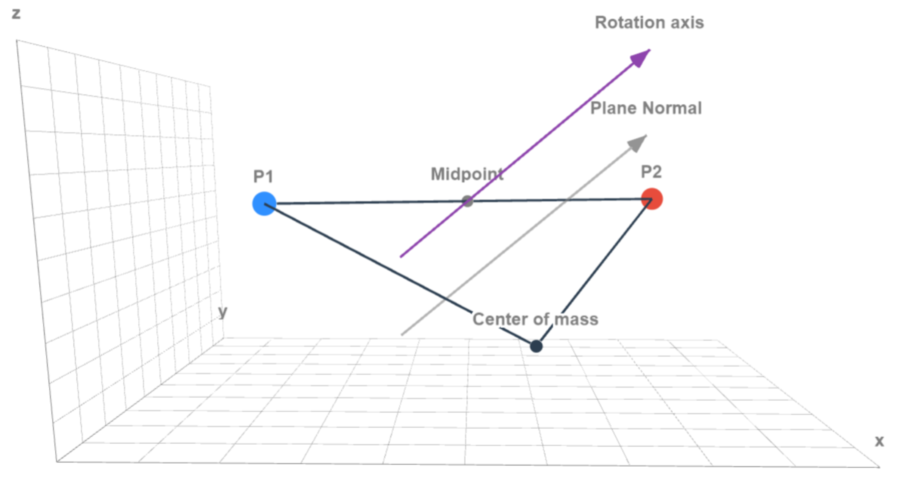


Figure 36: Rotation axis as an axis that goes through the midpoint of P_1 and P_2, and is parallel to the normal vector of the plane formed by P_1, P_2 and the center of mass.

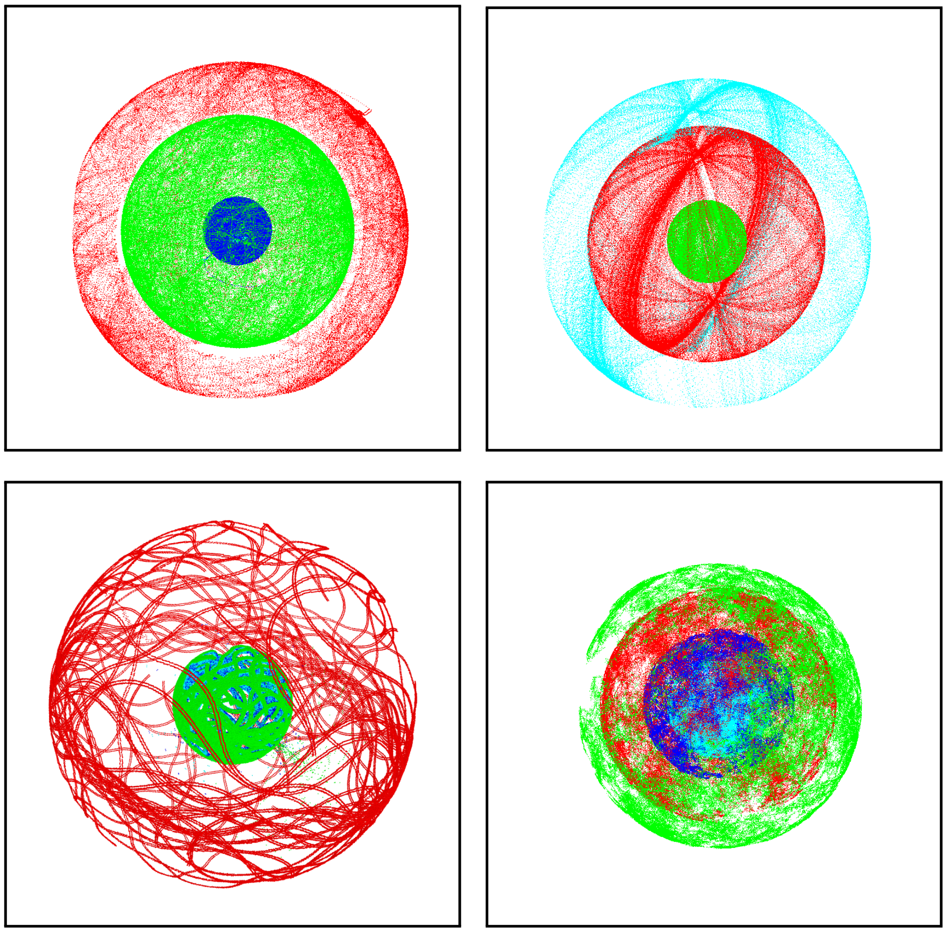


Figure 37: 3D-Spiroplots generated by the fourth rotation method.

4.2 Properties of 3D-Spiroplots

For all our designed 3D-Spiroplot rotation methods, with the exception of multi-axis rotation, we preserve the property of maintaining the center of mass of all active points after a rotation. This property is ensured because the chosen axis of rotation passes through the midpoint of the two active points of the current r-triplet. A formal proof for this conservation property is applicable to all methods except multi-axis rotation, by the following:

The center of mass of n points is calculated by taking the average of their coordinates. Suppose the center of mass $C = (\overline{x}, \overline{y}, \overline{z})$, with $\overline{x} = \frac{x_1+x_2+...+x_n}{n}, \overline{y} = \frac{y_1+y_2+...+y_n}{n}$ and $\overline{z} = \frac{z_1+z_2+...+z_n}{n}$. The formula of calculating the x-coordinate of the center of mass can be rewritten to:

$$\begin{aligned}
\frac{x_1 + x_2 + ... + x_n}{n} &= \frac{x_1 + x_2 + ... + x_{n-2}}{n} + \frac{x_{n-1} + x_n}{n} && (1)\\
&= \frac{x_1 + x_2 + ... + x_{n-2}}{n} + \frac{\frac{x_{n-1}+x_n}{2} * 2}{n} && (2)\\
&= \frac{x_1 + x_2 + ... + x_{n-2}}{n} + \frac{x_{n-1} + x_n}{2} * \frac{2}{n} && (3)
\end{aligned}$$

The same rewrite steps can be performed in the the formula of calculating the y and z-coordinates of the center of mass. The benefit of this rewritten formula is that we isolate two points from the rest, to calculate center of mass of all points. When only the coordinates of x_{n-1} and x_n change, only the following part of the formula has to be recalculated: $\frac{x_{n-1}+x_n}{2}$, since n and the values of x_1 to x_{n-2} remains unchanged.

For the proof, consider a 3D-Spiroplot configuration with n points. In each iteration we set the first point of the current r-triplet as $P_{n-1} = (x_{n-1}, y_{n-1}, z_{n-1})$ and the second point as $P_n = (x_n, y_n, z_n)$, all other points can be arbitrarily set from $P_1 = (x_1, y_1, z_1)$ to $P_{n-2} = (x_{n-2}, y_{n-2}, z_{n-2})$. With the previous formula we now only have to recalculate $\frac{x_{n-1}+x_n}{2}$, $\frac{y_{n-1}+y_n}{2}$ and $\frac{z_{n-1}+z_n}{2}$, which is the same as calculating the new midpoint of P_n and P_{n-1}, to obtain the new center of the mass after rotation. To prove if the center of mass remains unchanged after rotation for the specified rotation methods, we now only need to prove that the midpoint of the points of the current r-triplet stays the same after a rotation.

Let $A = (x_1, y_1, z_1)$ and $B = (x_2, y_2, z_2)$ be the original coordinates of two points in 3D space, and let their midpoint be M. The coordinates of M are given by:

$$M = \left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}, \frac{z_1 + z_2}{2}\right)$$

Now, let's rotate A and B by an angle θ around a common axis passing through M. Let the coordinates of the rotated points be $A' = (x_1', y_1', z_1')$ and $B' =$

(x_2', y_2', z_2'). A 3D rotation around the origin can be represented as a matrix operation on the coordinates of the points. Specifically, we can write:

$$\mathbf{A}' = R\mathbf{A} \quad \text{and} \quad \mathbf{B}' = R\mathbf{B}$$

where R is the 3x3 rotation matrix corresponding to the axis and angle θ. To rotate A and B around M, we first translate the points so that M coincides with the origin, then apply the rotation, and finally translate back.

$$\mathbf{A'_m} = R(\mathbf{A} - \mathbf{M}) + \mathbf{M} \quad \text{and} \quad \mathbf{B'_m} = R(\mathbf{B} - \mathbf{M}) + \mathbf{M}$$

The new midpoint M' of A'_m and B'_m is given by:

$$\mathbf{M}' = \frac{1}{2}(\mathbf{A'_m} + \mathbf{B'_m})$$

With a couple of rewriting steps, we find:

$$\begin{aligned}
\mathbf{M}' &= \frac{1}{2}\left(R(\mathbf{A} - \mathbf{M}) + \mathbf{M} + R(\mathbf{B} - \mathbf{M}) + \mathbf{M}\right) \\
&= \frac{1}{2}\left(R\mathbf{A} - R\mathbf{M} + \mathbf{M} + R\mathbf{B} - R\mathbf{M} + \mathbf{M}\right) \\
&= \frac{1}{2}\left(R\mathbf{A} + R\mathbf{B}\right) + \frac{1}{2}\left(\mathbf{M} + \mathbf{M} - R\mathbf{M} - R\mathbf{M}\right) \\
&= \frac{1}{2}\left(R\mathbf{A} + R\mathbf{B}\right) + \mathbf{M} - R\mathbf{M}
\end{aligned}$$

Since $\mathbf{M} = \frac{1}{2}(\mathbf{A} + \mathbf{B})$, it follows that $R\mathbf{M} = \frac{1}{2}R(\mathbf{A} + \mathbf{B}) = \frac{1}{2}(R\mathbf{A} + R\mathbf{B})$. Therefore, we have:

$$\mathbf{M}' = \frac{1}{2}\left(R\mathbf{A} + R\mathbf{B}\right) + \mathbf{M} - R\mathbf{M} = \frac{1}{2}\left(R\mathbf{A} + R\mathbf{B}\right) + \mathbf{M} - \frac{1}{2}\left(R\mathbf{A} + R\mathbf{B}\right) = \mathbf{M}$$

This completes the proof that the midpoint M' after the rotation is the same as the original midpoint M, as well as that the center of mass is maintained after a rotation for the specified rotation methods.

For multi-axis rotation we can prove that it does not preserve the center of mass by the following:

If the two points A and B are rotated around two different axes, both passing through their common midpoint M with the same angle θ, then the midpoint M' after rotation will generally not be the same as the original midpoint M. Let's consider R_1 and R_2 be the rotation matrices for the two different axes passing through M. For the two points A and B, their new positions A' and B' after rotation will be:

$$\mathbf{A}' = R_1(\mathbf{A} - \mathbf{M}) + \mathbf{M}$$

$$\mathbf{B}' = R_2(\mathbf{B} - \mathbf{M}) + \mathbf{M}$$

The new midpoint M' of A' and B' would then be:

$$\begin{aligned} M' &= \frac{1}{2}\left(\mathbf{A}' + \mathbf{B}'\right) \\ &= \frac{1}{2}\left(R_1(\mathbf{A} - \mathbf{M}) + \mathbf{M} + R_2(\mathbf{B} - \mathbf{M}) + \mathbf{M}\right) \\ &= \frac{1}{2}(R_1\mathbf{A} + R_2\mathbf{B}) + \frac{1}{2}\left(\mathbf{M} + \mathbf{M} - R_1\mathbf{M} - R_2\mathbf{M}\right) \end{aligned}$$

The new midpoint M' will depend on the rotated positions $R_1\mathbf{A}$ and $R_2\mathbf{B}$, as well as $R_1\mathbf{M}$ and $R_2\mathbf{M}$, which are generally not the same as $\mathbf{A}$, $\mathbf{B}$ or $\mathbf{M}$ unless $R_1 = R_2$ (i.e., the axes of rotation are the same). We demonstrate this using an example:

Let $A = (0, 4, 0), B = (2, 2, 0)$ and $C = (-2, -6, 0)$. By averaging the coordinate position of the points we calculate that the center of mass of these points lies on the origin $(0, 0, 0)$. Suppose the current r-triplet $R = (A, B, 90)$, using the multi-axis rotation method point A will rotate around the axis that goes through the midpoint M of A and B and is parallel to the unrotated position vector of B, and vice versa. We can represent the rotations as matrix operation on the coordinates of the points:

$$\mathbf{A}' = R_1(\mathbf{A} - \mathbf{M}) + \mathbf{M} \quad \text{and} \quad \mathbf{B}' = R_2(\mathbf{B} - \mathbf{M}) + \mathbf{M}$$

where $M = (1, 3, 0)$ and, R_1 and R_2 are the 3x3 rotation matrices corresponding to the axis and angle $\theta = 90°$. After normalizing the position vectors of A and B we get $(0, 1, 0)$ and $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0)$ respectively. The general rotation matrix R about a unit vector $k = (k_x, k_y, k_z)$ by angle θ is:

$$R = \begin{bmatrix} \cos(\theta) + k_x^2(1 - \cos(\theta)) & k_x k_y(1 - \cos(\theta)) - k_z \sin(\theta) & k_x k_z(1 - \cos(\theta)) + k_y sin(\theta) \\ k_y k_x(1 - \cos(\theta)) + k_z \sin(\theta) & \cos(\theta) + k_y^2(1 - \cos(\theta)) & k_y k_z(1 - \cos(\theta)) - k_x \sin(\theta) \\ k_z k_x(1 - \cos(\theta)) - k_y sin(\theta) & k_z k_y(1 - \cos(\theta)) + k_x \sin(\theta) & \cos(\theta) + k_z^2(1 - \cos(\theta)) \end{bmatrix}$$

By plugging in the values for R_1 and R_2 we get:

$$R_1 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{\sqrt{2}} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{2} \end{bmatrix} \quad \text{and} \quad R_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

The rotated coordinate positions of A and B are:

$$\mathbf{A}' = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{\sqrt{2}} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 0 - 1 \\ 4 - 3 \\ 0 - 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ \frac{2}{\sqrt{2}} \end{bmatrix}$$

$$\mathbf{B}' = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2-1 \\ 2-3 \\ 0-0 \end{bmatrix} + \begin{bmatrix} 1 \\ 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}$$

The new center of mass after rotation of point A, B and C is $(0, -\frac{1}{3}, \frac{1}{3}(\sqrt{2}-1))$. This proves that the center of mass is not guaranteed to be maintained, when using multi-axis rotation.

4.3 3D-Spiroplot application

As previously mentioned, an application has been developed to generate and visualize 3D-Spiroplots. Each point plotted is stored within a cell of a 3D grid, which we present as an interactive voxel world equipped with a movable camera. To project the voxel onto the screen, we employed ray-tracing techniques. For enhanced efficiency, we integrated the Fast Voxel Traversal Algorithm for Ray Tracing, as detailed in the following source: `http://www.cse.yorku.ca/~amana/research/grid.pdf`

Furthermore, we used a three-level hierarchical grids structure for ray traversal. One advantage of this approach is that the space is subdivided into finer grids, establishing a hierarchical structure. As a result, a ray first intersects with the broader, more general grid. If an intersection with a cell containing a voxel is identified, it then progresses to the smaller, detailed grids. This method significantly reduces the number of cells that require examination when the grid is sparse. To enhance performance further, we have incorporated GPGPU code. The frameworks of OpenCL and OpenGL have been used to create a window scene and to display the voxels. Lastly, the 3D-Spiroplots can be exported as OBJ files, where each occupied cell is represented by six quads. The application can be downloaded from the following link: `https://gitlab.com/liyeun/Spiroplot.git`

5 Conclusion and Discussions

We presented a detailed exploration of the influence of various bit-precision levels on the Spiroplot's accuracy, revealing that double precision is typically sufficient for generating accurate Spiroplots for fewer than 10^{14} iterations. We further showcased the trade-offs between computational time and precision. Moreover, our enhancements to the original Spiroplot application have introduced additional methods for determining point-drawing order, enhancing its versatility. In the second half of the paper we have discussed and demonstrated the extension of the original 2D-Spiroplots to the three-dimensional realm. By presenting different methods of defining the rotation axis, we have been able to produce a range of intriguing 3D figures. All the discussed rotation methods, except for multi-axis rotation, conservatively maintain the center of mass of all active points post-rotation, which is a property carried over from the original 2D-Spiroplots.

Further research could delve deeper into optimizing the parameters and settings for generating 3D-Spiroplots of higher aesthetic appeal or understanding the mathematical properties that govern their formation in greater depth. Additionally, exploring alternative methods or combining different rotation methods might lead to even more complex and mesmerizing plots.

The potential applications of these 3D-Spiroplots extend beyond mere aesthetics, possibly serving as unique visualization tools for complex data sets, or as novel artistic patterns in design and architecture.

Appendices

A Image comparisons

The next eight figures present side-by-side comparisons of Spiroplot configurations after 10 million iterations. In each figure:

- The left image displays the Spiroplot generated using single precision.
- The center image showcases the Spiroplot produced using octuple precision.
- The right image highlights the differences between the two.

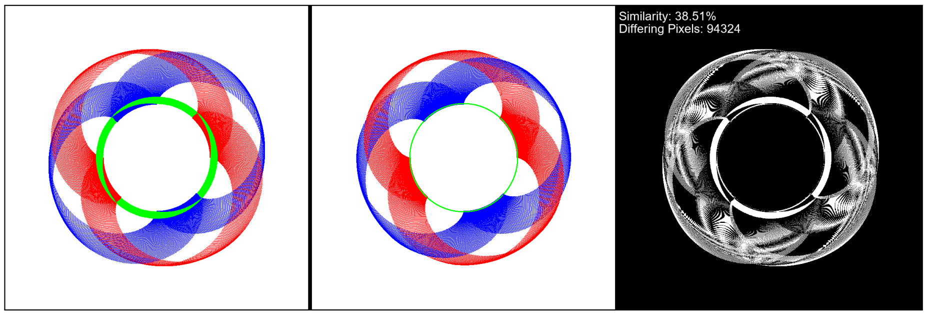


Figure 38: Second Spiroplot configuration of three initial points coordinates: $p_1 = (200, 500), p_2 = (500, 500)$ and $p_3 = (500, 200)$. With r-triplets $(p_1, p_2, 1)$ and $(p_2, p_3, 1)$.

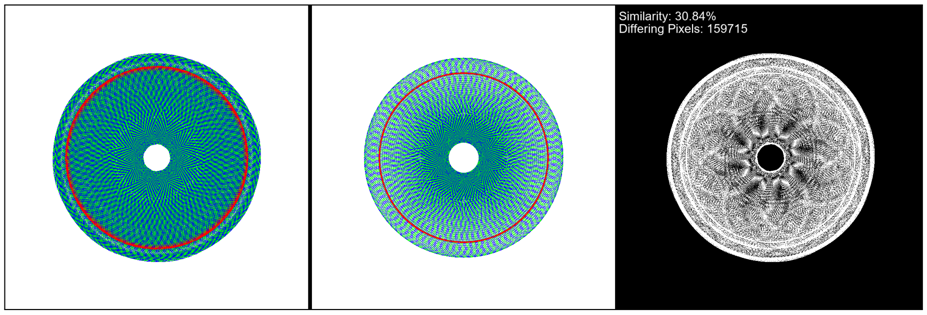


Figure 39: Third Spiroplot configuration of three initial points coordinates: $p_1 = (200, 500), p_2 = (500, 500)$ and $p_3 = (500, 200)$. With r-triplets $(p_1, p_2, 1), (p_2, p_3, 90)$ and $(p_1, p_3, 1)$.

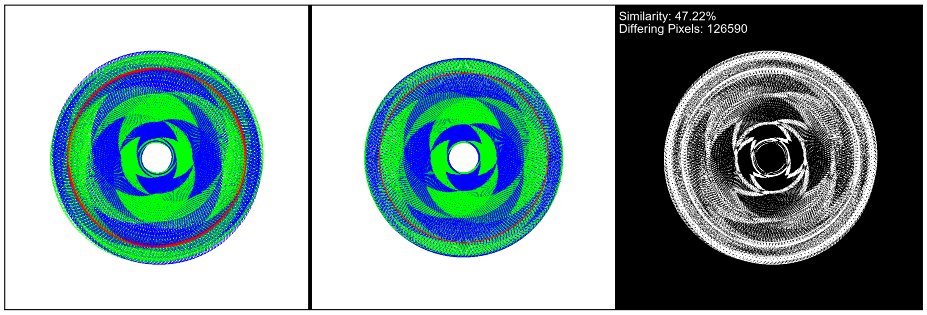


Figure 40: Fourth Spiroplot configuration of three initial points coordinates: $p_1 = (200, 500), p_2 = (500, 500)$ and $p_3 = (500, 200)$. With r-triplets $(p_3, p_2, 1), (p_2, p_1, 1), (p_3, p_2, 1)$ and $(p_3, p_1, 1)$.

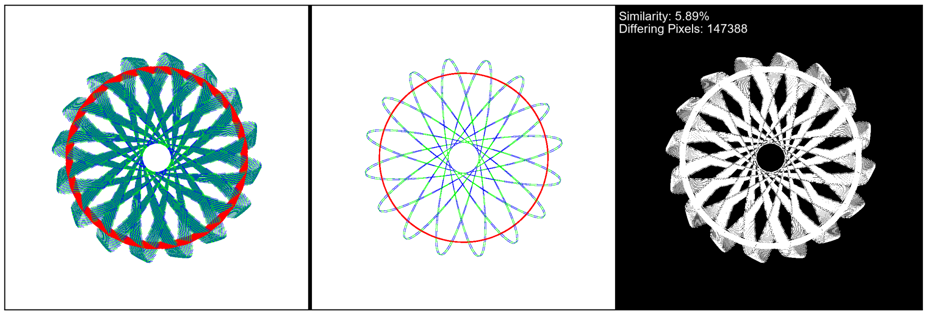


Figure 41: Fifth Spiroplot configuration of three initial points coordinates: $p_1 = (200, 500), p_2 = (500, 500)$ and $p_3 = (500, 200)$. With r-triplets $(p_1, p_2, 1), (p_2, p_3, 43)$ and $(p_3, p_1, 1)$.

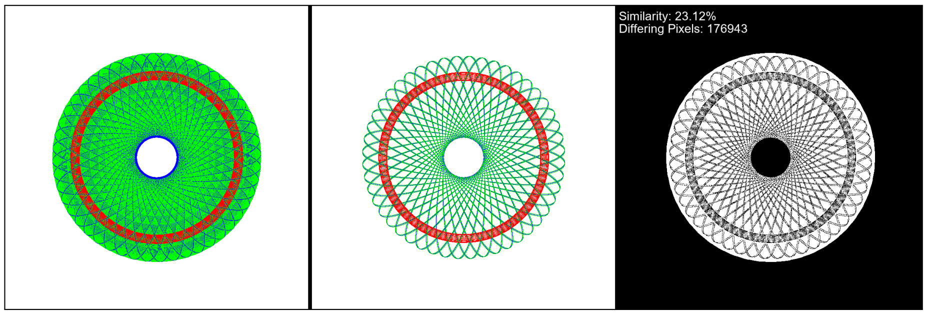


Figure 42: Sixth Spiroplot configuration of three initial points coordinates: $p_1 = (200, 500), p_2 = (500, 500)$ and $p_3 = (500, 200)$. With r-triplets $(p_1, p_2, 16)$ and $(p_2, p_3, 179)$.

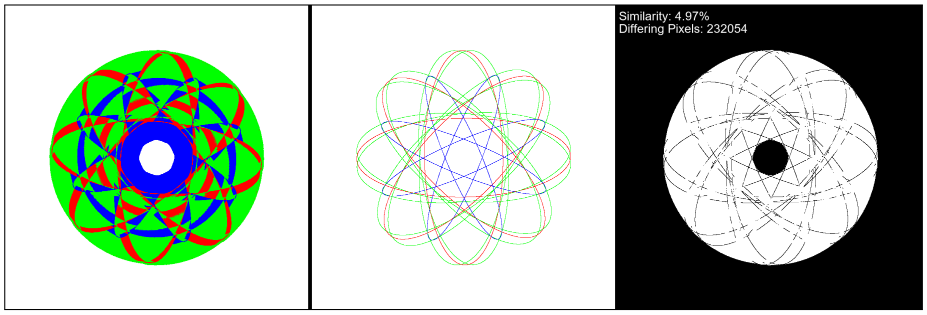


Figure 43: Seventh Spiroplot configuration of three initial points coordinates: $p_1 = (200, 500), p_2 = (500, 500)$ and $p_3 = (500, 200)$. With r-triplets $(p_1, p_2, 172)$ and $(p_2, p_3, 98)$.

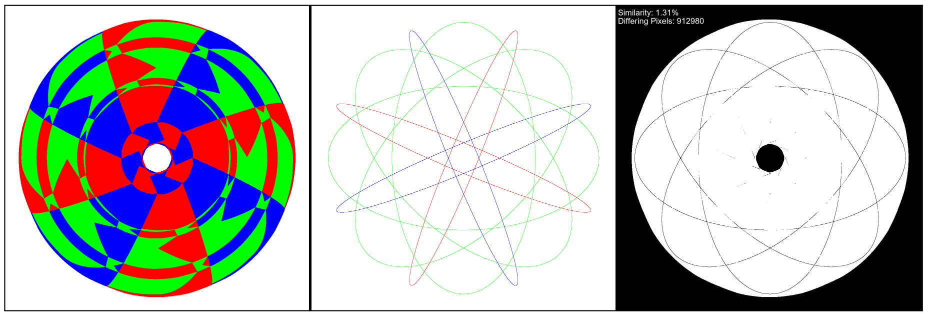


Figure 44: Eighth Spiroplot configuration of three initial points coordinates: $p_1 = (0, 600), p_2 = (600, 600)$ and $p_3 = (600, 0)$. With r-triplets $(p_1, p_2, 90)$ and $(p_2, p_3, 90)$.

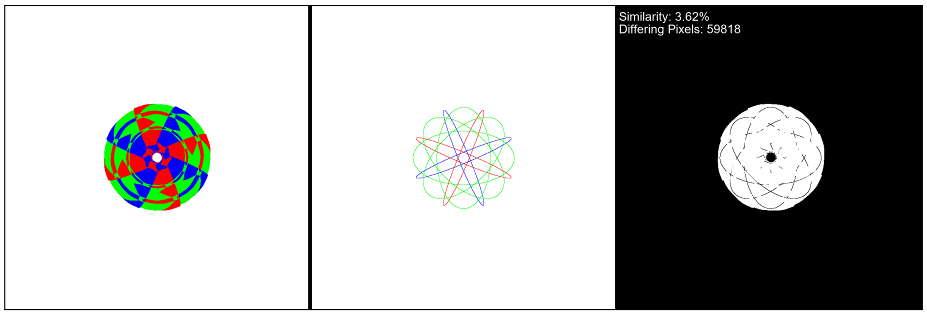


Figure 45: Ninth Spiroplot configuration of three initial points coordinates: $p_1 = (300, 450), p_2 = (450, 450)$ and $p_3 = (450, 300)$. With r-triplets $(p_1, p_2, 90)$ and $(p_2, p_3, 90)$.

References

[1] Andrew Adamatzky. *Game of Life Cellular Automata.* Springer, 2010. DOI: 10.1007/9781849962179.

[2] C. Alexander and M. Sadiku. *Fundamentals of Electric Circuits.* McGraw-Hill, 2012.

[3] Diogo de Andrade, Nuno Fachada, Carlos Fernandes, and Agostinho Rosa. “Generative Art with Swarm Landscapes”. In: *Entropy* 22 (Nov. 2020), p. 1284. DOI: 10.3390/e22111284.

[4] Michael F. Barnsley. *Fractals Everywhere.* Academic Press, 2014.

[5] G. Cagdas. “A shape grammar: The language of traditional Turkish houses”. In: *Environment and Planning B: Planning and Design* 23 (July 1996), pp. 443–464. DOI: 10.1068/b230443.

[6] M. H. Christensen. “Structural Synthesis using a Context Free Design Grammar Approach”. In: *Computer Science, Linguistics* (2009).

[7] A. T. Crooks and A. J. Heppenstall. “Introduction to Agent-Based Modelling”. In: *Agent-Based Models of Geographical Systems.* Ed. by A. Heppenstall, A. Crooks, L. See, and M. Batty. Dordrecht: Springer, 2012. DOI: 10.1007/978-90-481-8927-4_5.

[8] Robert Devaney. *A First Course in Chaotic Dynamical Systems: Theory and Experiment.* 2020. ISBN: 9780429280665. DOI: 10.1201/9780429280665.

[9] Casper van Dommelen, Marc van Kreveld, and Jérôme Urhausen. “Spiroplots: a New Discrete-time Dynamical System to Generate Curve Patterns”. In: *Proceedings of Bridges 2020: Mathematics, Art, Music, Architecture, Education, Culture.* 2020.

[10] L. Edelstein-Keshet. *Mathematical Models in Biology.* Random House-/Birkhäuser, 1988.

[11] Gilberto Espinosa-Paredes. “Fractional-order modeling in nuclear reactors”. In: Jan. 2021, pp. 1–39. ISBN: 9780128236659. DOI: 10.1016/B978-0-12-823665-9.00001-8.

[12] *Exploring Neural Differential Equations in Generative AI.* Accessed: 2023-20-09. 2023. URL: https://www.analyticsvidhya.com/blog/2023/08/neural-differential-equations-in-generative-ai/.

[13] D. Fisher. *Drawing apparatus.* U.S. Patent No. 3415303. 1965.

[14] Théotime Gros. “Can Artificial Intelligence Create Art?” PhD thesis. June 2019. DOI: 10.13140/RG.2.2.10238.41287.

[15] N. Gu and P. Amini Behbahani. “Shape Grammars: A Key Generative Design Algorithm”. In: *Handbook of the Mathematics of the Arts and Sciences.* Ed. by B. Sriraman. Cham: Springer, 2021. DOI: 10.1007/978-3-319-57072-3_7.

[16] Michel Hénon. “A two-dimensional mapping with a strange attractor”. In: *Communications in Mathematical Physics* 50 (1976), pp. 69–77.

[17] Mikael Hvidtfeldt Christensen. *Generative Art, 3D Fractals, Creative Computing.* Accessed: 2023-04-11. 2008. URL: `http://blog.hvidtfeldts.net/index.php/2008/12/grammars-for-generative-art-part-ii/`.

[18] Andrew Ilachinski. *Cellular automata: A discrete universe.* World Scientific, 2001.

[19] D. Jurafsky and J. H. Martin. *Speech and Language Processing.* Pearson, 2019.

[20] Ranjit Konkar. “Flattening the Curve. . . of Spirographs”. In: *Recreational Mathematics Magazine* 9 (2022), pp. 1–20. DOI: `10.2478/rmm-2022-0001`.

[21] Robert Krawczyk. *Spirolaterals, Complexity From Simplicity.* 2000.

[22] J. D. Lawrence. *A Catalog of Special Plane Curves.* New York: Dover, 1972, pp. 168–170.

[23] J. A. Lissajous. “Mémoire sur l’étude optique des mouvements vibratoires”. In: *Annales de chimie et de physique* 3 (1857), pp. 147–231.

[24] *Lissajous Curve Tracing Algorithm.* Accessed: 2023-04-11. 2019. URL: `https://www.101computing.net/python-turtle-lissajous-curve/`.

[25] Edward N. Lorenz. “Deterministic Nonperiodic Flow”. In: *Journal of the Atmospheric Sciences* 20.2 (1963), pp. 130–141.

[26] MacTutor History of Mathematics Archive. *Rose curves.* `https://mathshistory.st-andrews.ac.uk/Curves/Rhodonea/`. Accessed: 2023-04-11. University of St Andrews.

[27] Benoit B. Mandelbrot. *The Fractal Geometry of Nature.* Vol. 173. WH Freeman, 1983.

[28] J. McCormack and M. d’Inverno. *Computers and creativity.* Springer, 2012.

[29] D. McKenna. “From Lissajous to Pas de Deux to Tattoo: The Graphic Life of a Beautiful Loop”. In: *Proceedings of Bridges 2011: Mathematics, Music, Art, Architecture, Culture.* Ed. by R. Sarhangi and C. H. Séquin. Tessellations Publishing. Phoenix, Arizona, 2011, pp. 295–302.

[30] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and fractals: New frontiers of science.* Springer, 2004.

[31] Przemyslaw Prusinkiewicz. “Graphical applications of L-systems”. In: *Proceedings of Graphics Interface.* Vol. 86. 1986, pp. 247–253.

[32] Python Software Foundation. *turtle – Turtle graphics.* `https://docs.python.org/3/library/turtle.html`. Accessed: 2023-04-11.

[33] *Python Turtle Spirograph.* Accessed: 2023-04-11. 2018. URL: `https://www.101computing.net/python-turtle-spirograph/`.

[34] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. *Zero-shot Text-to-Image Generation*. 2021. DOI: `10.48550/arXiv.2102.12092`. arXiv: `2102.12092 [cs.CV]`.

[35] Craig W. Reynolds. "Flocks, herds and schools: A distributed behavioral model". In: *ACM SIGGRAPH Computer Graphics* 21.4 (1987), pp. 25–34.

[36] Larry Riddle. *Koch Snowflake*. Accessed: 2023-04-11. 2022. URL: `http://larryriddle.agnesscott.org/ifs/ksnow/ksnow.htm`.

[37] Steven H. Strogatz. *Nonlinear dynamics and chaos: With applications to physics, biology, chemistry, and engineering*. Westview Press, 2014.

[38] *Vera Molnár's Artwork revisited using Python*. Accessed: 2023-04-11. 2020. URL: `https://www.101computing.net/vera-molnar-artwork-revisited-using-python/`.

[39] Eric W. Weisstein. *Epitrochoid*. MathWorld–A Wolfram Web Resource. Accessed: 2023-04-11. URL: `https://mathworld.wolfram.com/Epitrochoid.html`.

[40] Eric W. Weisstein. *Hypotrochoid*. MathWorld–A Wolfram Web Resource. Accessed: 2023-04-11. URL: `https://mathworld.wolfram.com/Epitrochoid.htmll`.

[41] Eric W. Weisstein. *Rose Curve*. MathWorld–A Wolfram Web Resource. Accessed: 2023-04-11. URL: `https://mathworld.wolfram.com/RoseCurve.html`.

[42] Haoran Wen. "A review of the Hénon map and its physical interpretations". In: (Jan. 2014).

[43] F. M. White. *Viscous Fluid Flow*. McGraw-Hill, 2006.

[44] H. Irwine Whitty. *The Harmonograph*. Norwich: Jarrold & Sons, 1893.

[45] Rick Wickling. *Iterated function systems and Barnsley's fern in SAS*. Accessed: 2023-04-11. 2012. URL: `https://blogs.sas.com/content/iml/2012/12/12/iterated-function-systems-and-barnsleys-fern-in-sas.html`.