Differentiable Iterated Function Systems

Cory B. Scott [1]

Abstract

This preliminary paper presents initial explorations in rendering Iterated Function System (IFS) fractals using a differentiable rendering pipeline. Differentiable rendering is a recent innovation at the intersection of computer graphics and machine learning. A fractal rendering pipeline composed of differentiable operations opens up many possibilities for generating fractals that meet particular criteria. In this paper I demonstrate this pipeline by generating IFS fractals with fixed points that resemble a given target image - a famous problem known as the *inverse IFS problem*. The main contributions of this work are as follows: 1) I demonstrate (and make code available for) this rendering pipeline; 2) I discuss some of the nuances and pitfalls in gradient-descent-based optimization over fractal structures; 3) I discuss best practices to address some of these pitfalls; and finally 4) I discuss directions for further experiments to validate the technique.

1. Introduction and Prior Work

Fractals are ubiquitous objects in computer graphics, mathematical art, and data analysis. A common way to generate fractal images is the *iterated function system* (IFS), defined in detail below. In this paper, I demonstrate a differentiable system for drawing IFS fractals written in Pytorch. The advantage of doing this in Pytorch is that it enables optimization through the fractal rasterization process, so that the generated IFS fractal resembles a target image. This paper is inspired by the recent work "Differentiable Drawing and Sketching", by Mihai et al. [1]. Those authors presented a framework for differentiating through the rasterization of lines and curves, using the insight that the distance from a pixel to a line can act as a differentiable proxy for rasterization. Since distance estimators have a long history of use in the fractal rendering community [2]–[4], it is natural to combine some of those ideas with the techniques from Mihai et al. I will first define some of the background needed to understand the rendering problem, and then I will introduce how the rendering pipeline works. Throughout this paper, I will use the Koch curve (a famous fractal [5]) as a test case.

1.1. IFS Fractals

I follow the definition of IFS fractal given by Barnsley [6] and later by Heptig [7]. Let $F_1, F_2 \ldots F_k$ be a set of affine transformations of the Euclidean plane $\mathbb{R}^2$, and for any region $\mathcal{S} \subset \mathbb{R}^2$ let $F_i(\mathcal{S})$ denote the image of $\mathcal{S}$ under F_i. The *Hutchinson* transform [8] $H(S)$ is the union of the images of $\mathcal{S}$ under all of the F_i: $H(\mathcal{S}) = \cup_{i=1}^{n} F_i(\mathcal{S})$.

The *attractor* of an IFS is a limit figure (not necessarily unique) which results from applying this process infinitely many times, i.e. a region A such that $A = H(A)$. For an attractor to exist, the set of affine transformations must on average be *contractive*, meaning that they shrink the distance between points in the plane.

A related common construction for self-similar curves is to start with a base figure, and then recursively replace parts of the current figure with a transformed copy of the entire figure. For example, the Koch curve (see Figure 1) is generated by beginning with the line segment between $(0,0)$ and $(0,1)$ and recursively applying the four affine transformations shown in Equation 1.

$$\begin{aligned} T_1 &= \begin{bmatrix} \frac{1}{3} & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & 1 \end{bmatrix} T_2 = \begin{bmatrix} \frac{1}{6} & -\frac{\sqrt{3}}{6} & \frac{1}{3} \\ \frac{\sqrt{3}}{6} & \frac{1}{6} & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ T_3 &= \begin{bmatrix} \frac{1}{6} & \frac{\sqrt{3}}{6} & \frac{1}{3} \\ -\frac{\sqrt{3}}{6} & \frac{1}{6} & 0 \\ 0 & 0 & 1 \end{bmatrix} T_4 = \begin{bmatrix} \frac{1}{3} & 0 & \frac{2}{3} \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{1}$$

1.2. IFS Inverse Problem

Now that I have defined IFS fractals, I am ready to introduce the *inverse IFS problem*. Simply put, this is the problem

[1]Department of Mathematics and Computer Science, Colorado College, Colorado Springs, USA. Correspondence to: Cory B. Scott <cbs@coloradocollege.edu>.

Published at the 2^{nd} Differentiable Almost Everything Workshop at the 41^{st} International Conference on Machine Learning, Vienna, Austria. July 2024. Copyright 2024 by the author(s).

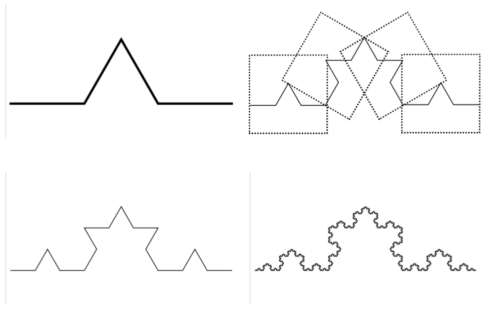

Figure 1. From left to right: First row: The first iteration of the Koch IFS fractal; the second iteration, showing its composition as several scaled and rotated copies of the first iteration. Second row: As in the second image, but without bounding boxes; the attractor of this process.

of finding a set of affine transformations having a specific image as their attractor. This problem is also called the "fractal image compression" problem, motivated by storing a set of affine transformations instead of the pixels of the original image. The inverse IFS problem is a classic problem, and a wide variety of approaches have been proposed. An early approach due to Barnsley et. al [9] proposed optimizing a loss function that measured distortion of the target region under the Hausdorff metric ("the Collage Theorem"). Other approaches have been proposed which utilize the convex hull of the target region [10]; genetic algorithms [11]; wavelet transforms [12]; EM-like algorithms [13]; and in some cases, neural networks [14], [15]. However, no general solution is known to exist. Prior work by Melnik et. al [16], similar to this paper, uses gradient descent to optimize IFS fractals. Those authors train a recurrent neural network $R(X)$ to iteratively transform a set of points X, with the Hausdorff metric between X and $R(X)$ as the loss function. In contrast to this paper, Melnik et. al's work does not generate an explicit set of affine transformations. Smooth representations of IFS fractals (and fitting IFS fractals to images) have also been considered as one application of differentiable programming by Petersen et al. [17]. Poli et al. [18] consider neural networks that learn affine transformations, which they call a "neural collage", after Barnsley's approach. I stress here that I do not claim to have solved the inverse IFS problem in this paper; As in Petersen et. al, I am using this problem to demonstrate the utility of differentiating through fractal rasterization.

1.3. Signed Distance Functions

Signed distance functions are a way to represent the geometry of a region in Euclidean space. An SDF is the distance between a point in space and the boundary of a shape; points inside the shape are assigned the negative of this distance. SDFs have several properties that make them attractive in a rendering context: 1) for many geometric primitives exact SDF formulae are known [19]; 2) in Euclidean space, the SDF of a region is differentiable almost everywhere; and 3) SDFs compose nicely with each other (for example, in Boolean operations) and with affine transformations. These properties (amongst others) have lead to widespread integration of SDFs into a variety of machine learning approaches in recent years [20]–[25].

One specific characteristic of SDFs that will be necessary later in this paper is how a SDF behaves under affine transformation. Let $d_{\mathcal{S}}(x)$ be a SDF for a region $\mathcal{S}$, and let $T(x)$ be an affine transformation with scale parameter s. Then

$$d_{T(\mathcal{S})}(x) = \frac{1}{s} d(T^{-1}(x)) \tag{2}$$

In other words, one can easily calculate a new SDF that represents an affinely transformed version of $\mathcal{S}$ by evaluating d on x but with the inverse transform of T. Note that this only holds when T scales space uniformly - this equation does not hold for shear deformations.

1.4. Differentiable Rendering

Automatic Differentiation ("autodiff" or AD) is a programming paradigm that has aided in the explosive growth of deep learning. The basic idea of autodiff is that all basic operations in the programming language are defined in a way that includes their derivative. This means that it is possible to use the chain rule to compute the derivative of an entire program with respect to its inputs, enabling optimization via gradient descent. Recent work has successfully used autodiff to write full rasterization pipelines which are differentiable. Mihai et al. [1] detail how to (differentiably) turn an implicit line segment into a pixelated picture of a line segment; in the next section I take the ability to do this as a given, and use it to produce fractal images.

Some recent work has successfully used autodiff and signed distance functions to write full rasterization and rendering pipelines which are differentiable. The key insight is that a geometric primitive P can be drawn by coloring pixels according to their distance from the primitive [1]. To render the pixel at location (i, j), I take the exponential of the distance between the point (i, j) and the primitive: $\text{pixel}_{i,j} = \exp(-d((i,j),P)^2/\sigma^2)$. Because this pixel value function is a differentiable function of distance, and distance is a differentiable function of the parameters of P, I can tune these parameters using gradient descent. This allows fitting geometric primitives to images.

2. Rendering Process

The components of my proposed differentiable fractal are

- **Control points:** a set of variables $\mathcal{P} = \{p_1, p_2 \ldots p_n\}$, where all $p_i \in \mathbb{R}^2$.

- **Symmetry pattern:** a sequence L of pairs of subsets of $\mathcal{P}$: $L = \{(P_1, R_1), (P_2, R_2), \ldots (P_m, R_m)\}$ with all $P_i, R_i \subset \mathcal{P}$.
- **SDFs:** A set $\mathcal{D}$ of SDFs $d_1 \ldots d_l$, which may or may not also be defined in terms of the points p_i.

Each of these components could be specified in advance, or could be learned. For the remainder of this paper I will assume both the symmetry pattern and the SDFs are prespecified, and that the elements of $\mathcal{P}$ are parameters (tunable via gradient descent). However, it may be possible to also learn the symmetry pattern, or to use neural SDFs [20]. See Supplementary Material Section C for details.

To make the above more specific, the Koch curve could be parametrized with a set of 5 control points $\mathcal{P} = \{p_0, p_1, p_2, p_3, p_4\}$. To get the same self-similar behavior as the Koch curve, I would take the symmetry pattern as:

$$L = \{(\{p_0, p_4\}, \{p_0, p_1\}), (\{p_0, p_4\}, \{p_1, p_2\}), \\ (\{p_0, p_4\}, \{p_2, p_3\}), (\{p_0, p_4\}, \{p_3, p_4\})\}$$

In other words, L is specifying that the line segment $\{p_0, p_4\}$ should be transformed into the line segment $\{p_i, p_{i+1}\}$ via an affine transformation T_i, for each of $i = 0, 1, 2, 3$. The SDFs d_i would be functions representing the distance to each of these line segments (a closed formula for this distance is defined in terms of the endpoints of each segment). This choice of $(\mathcal{P}, L, \mathcal{D})$ has the same replacement logic as the Koch curve, making it possible (assuming all intermediate operations are differentiable) to optimize the locations of the p_i so that the linear transformations T_k match those in Equation 1. Specifying the self-similarity of the fractal in terms of which line segments get mapped to other line segments is general enough to describe a wide variety of fractal images; for another example (the symmetry pattern for a Sierpiński carpet), see Section B of the supplementary material.

Once the above components are defined, the process in generating a fractal figure is as follows:

1. For each i in $1 \ldots m$, find the linear transformation T_i which (possibly approximately) maps P_i to R_i.
2. Up to a set maximum number of recursions K, replace the SDFs in $\mathcal{D}$ with the result of applying each transformation T_i to all of the current elements of $\mathcal{D}$.
3. Compute the distance $d((i,j))$ from each pixel (i,j) to each SDF $d \in \mathcal{D}$. Color the pixel (i,j) with the value $\exp(-(\min_{d \in \mathcal{D}} d((i,j)))^2/\sigma^2)$. σ is a scale parameter that determines the ratio of pixel coordinates to the coordinates of the p_i.
4. (If optimizing) compare to target and compute loss.

To get an IFS fractal image that looks like a source image, I implemented the above steps in Pytorch. The position of the control points was tuned with Adam [26] to minimize the loss, i.e. so that the rendered image matches some target image. The maximum recursion depth, K, was set to be as deep as possible within GPU memory constraints. I will now discuss some of the details of the rendering pipeline; readers who want to recreate the images in this paper should refer to the Github repository. Figure 2 demonstrates the result of optimizing with different symmetry patterns, such as the replacement rule that yields the Sierpiński carpet. Figure 2 also includes the result of applying this optimization procedure to images that have the "wrong" symmetry pattern; that is, where L does not actually match any symmetries in the target image.

2.1. Implementation Details

In this section I describe several of the specific design choices I made in my fractal renderer.

Initial Conditions. As described, this system is extremely sensitive to initial conditions, which limits its applicability. For the Koch curve, I was able to consistently get optimization to converge by initializing the control points p_i along a best-fit line of the black pixels in the target image.

Calculating Transformations. An important ingredient in the above process is the calculation of the linear transformation which takes the endpoints to each line segment. To do this for two line segments (e_1, e_2) and (p_1, p_2), I: a) find the translation that takes e_1 to p_1, b) find the rotation that aligns the two vectors, and then c) scale (e_1, e_2) to have the same length as (p_1, p_2). The composition of these three linear transformations is the one I want.

Loss Function. Following the example of [1], I used Multiscale Mean Squared Error (MMSE) as the loss function. MMSE is identical to mean-squared error, but is summed over multiple pooled copies of the image. I experimented with two variants of loss function: computing MMSE over the pixel values in rasterized images, versus the MMSE between the raw distance values calculated by the SDF. Both of these loss functions worked reasonably well, but tended to get stuck in local optima (see Figure 4 for examples). Figure 3 illustrates why this might be happening: even arbitrarily close to the boundary of the fractal figure, the gradient of the loss (in this case, pixel loss) can point away from the location of the true optimum. Note in this image that the green arrows, representing MMSE, point toward the true optimum slightly more often than MSE at only the finest scale (in blue).

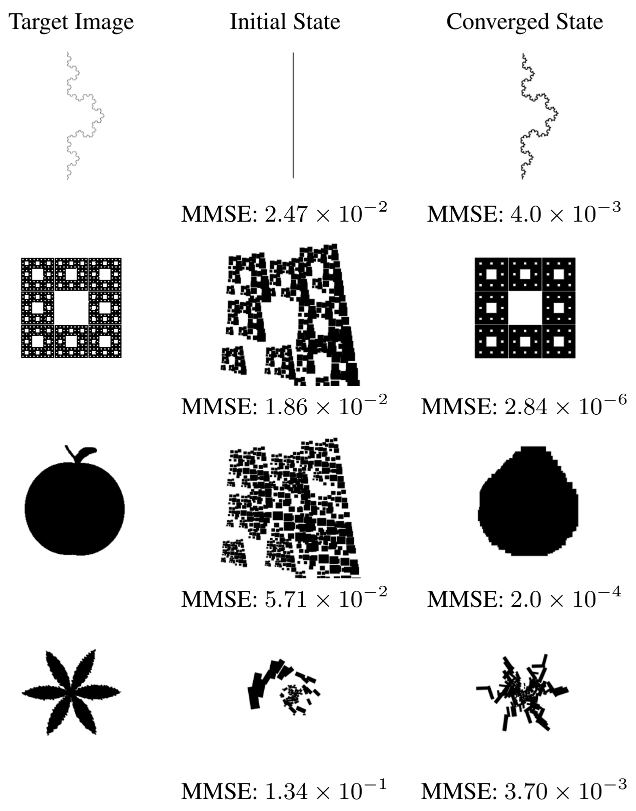
Target Image
Initial State
Converged State
MMSE: 2.47×10^{-2}
MMSE: 4.0×10^{-3}
MMSE: 1.86×10^{-2}
MMSE: 2.84×10^{-6}
MMSE: 5.71×10^{-2}
MMSE: 2.0×10^{-4}
MMSE: 1.34×10^{-1}
MMSE: 3.70×10^{-3}

Figure 2. Multiple examples of learning IFS attractors. In each row from top to bottom: the Koch curve; the Sierpiński carpet; an apple from the MPEG-7 shape dataset; a flower from the MPEG-7 shape dataset. MMSE values are given between each generated image and the target images.

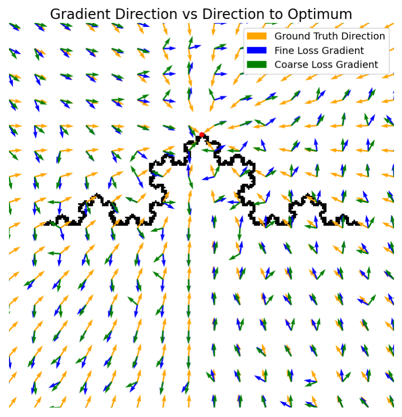

Figure 3. Vector fields illustrating the gradient of error as p_2 is varied in the Koch curve construction, while all other points are held constant at their optimal positions: a) always pointing toward the optimal location (orange); b) the gradient of the fine-scale loss only; and c) the gradient of the multiscale loss. While both gradient vector fields are divergent, the multiscale loss is slightly more aligned with the always-optimal field.

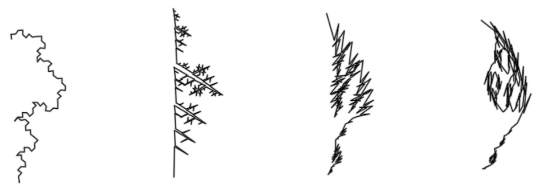

Figure 4. Examples of local optima encountered while optimizing a set of affine transformations to fit the Koch curve.

Code Repository All code for the operations described in this paper is available at `https://github.com/cory-b-scott/diff_ifs`.

3. Conclusion and Future Work

This paper presents initial evidence that it is possible to find the parameters of IFS fractals and self-similar curves using automatic differentiation and gradient descent. However, there are many open questions that still need to be addressed in order to make this a viable technique for solving the IFS inverse problem. In Figure 2, the Sierpiński and Koch examples both use the known, correct symmetry pattern for these IFSs. This is somewhat unfair, since learning the symmetry pattern in a domain is a harder problem than finding an affine transformation between point sets. Initial attempts at learning a symmetry pattern as a weighted combination of control points proved unsuccessful (optimization diverged in every case). One of the other IFS solving approaches may help here, as might recent machine learning work in automated discovery of symmetry in geometric objects [27].

Additionally, it is currently unclear what determines whether optimization will converge, diverge, or converge to a local minimum. Very careful initialization of control point locations is necessary to get results that resemble the target image. Developing a procedure for appropriately initializing control points given a target image is necessary to apply this technique to arbitrary images. The examples in Figure 4 seem to indicate that local minima are related to times when the figure crosses itself; a barrier function that prevents self-crossings could keep the optimization from getting trapped in these local optima.

Finally, there are also several modifications that could be made to make the process more efficient, such as the rendering tricks mentioned in [7]. One major area for future work is investigating the viability of coarse-to-fine optimization procedures, which are a mainstay of many graphics algorithms and optimization procedures.

Overall, this paper represents a proof-of-concept of differentiating through fractal rasterization. More work is necessary to quantitatively evaluate the proposed approach.

References

[1] D. Mihai and J. Hare, “Differentiable drawing and sketching,” *arXiv preprint arXiv:2103.16194*, 2021.

[2] M. H. Christensen, “Distance estimated 3d fractals,” 2011. [Online]. Available: http://blog.hvidtfeldts.net/index.php/2011/06/distance-estimated-3d-fractals-part-i/ (visited on 01/12/2022).

[3] J. C. Hart, D. J. Sandin, and L. H. Kauffman, “Ray tracing deterministic 3-d fractals,” in *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, 1989, pp. 289–296.

[4] M. McGuire, “Numerical methods for ray tracing implicitly defined surfaces,” *Williams College*, 2014.

[5] H. Koch, “Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire,” *Arkiv for Matematik, Astronomi och Fysik*, vol. 1, pp. 681–704, 1904.

[6] M. Barnsley and H. Rising, *Fractals Everywhere*. Elsevier Science, 1993, ISBN: 9780120790692. [Online]. Available: https://books.google.com/books?id=oh7NoePgmOIC.

[7] D. Hepting, P. Prusinkiewicz, and D. Saupe, “Rendering methods for iterated function systems,” in North-Holland, 1991.

[8] J. Hutchinson, “Fractals and self-similarity,” *Indiana Univ. Math. J.*, vol. 30, pp. 713–747, 5 1981, ISSN: 0022-2518.

[9] M. F. Barnsley, V. Ervin, D. Hardin, and J. Lancaster, “Solution of an inverse problem for fractals and other sets,” *Proceedings of the National Academy of Sciences*, vol. 83, no. 7, pp. 1975–1977, 1986.

[10] E. Hocevar, “An algorithm to solve the inverse ifs-problem,” Jan. 2002. DOI: 10.1007/978-1-4612-0089-5_40.

[11] E. Lutton, J. L. Véhel, G. Cretin, P. Glevarec, and C. Roll, “Mixed ifs: Resolution of the inverse problem using genetic programming,” Ph.D. dissertation, INRIA, 1995.

[12] R. Rinaldo and A. Zakhor, “Inverse and approximation problem for two-dimensional fractal sets,” *IEEE Transactions on Image Processing*, vol. 3, no. 6, pp. 802–820, 1994.

[13] P. Bloem and S. de Rooij, “An expectation-maximization algorithm for the fractal inverse problem,” *arXiv preprint arXiv:1706.03149*, 2017.

[14] D. La Torre, E. Maki, F. Mendivil, and E. Vrscay, “Iterated function systems with place-dependent probabilities and the inverse problem of measure approximation using moments,” *Fractals*, vol. 26, no. 05, p. 1 850 076, 2018.

[15] L. Graham and M. Demers, “Applying neural networks to a fractal inverse problem,” in *Recent Developments in Mathematical, Statistical and Computational Sciences: The V AMMCS International Conference, Waterloo, Canada, August 18–23, 2019*, Springer, 2021, pp. 157–165.

[16] O. Melnik and J. Pollack, “A gradient descent method for a neural fractal memory,” in *1998 IEEE International Joint Conference on Neural Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No. 98CH36227)*, IEEE, vol. 2, 1998, pp. 1069–1073.

[17] F. Petersen, C. Borgelt, and O. Deussen, “Algonet: C^∞ Smooth algorithmic neural networks,” *arXiv preprint arXiv:1905.06886*, 2019.

[18] M. Poli, W. Xu, S. Massaroli, C. Meng, K. Kim, and S. Ermon, “Self-similarity priors: Neural collages as differentiable fractal representations,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 30 393–30 405, 2022.

[19] I. Quilez, *Distance functions*, https://iquilezles.org/articles/distfunctions/, Accessed: 2024-06-02.

[20] J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove, “Deepsdf: Learning continuous signed distance functions for shape representation,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 165–174.

[21] D. Vicini, S. Speierer, and W. Jakob, “Differentiable signed distance function rendering,” *ACM Transactions on Graphics (TOG)*, vol. 41, no. 4, pp. 1–18, 2022.

[22] S. Osher, R. Fedkiw, S. Osher, and R. Fedkiw, “Constructing signed distance functions,” *Level set methods and dynamic implicit surfaces*, pp. 63–74, 2003.

[23] V. Sitzmann, E. Chan, R. Tucker, N. Snavely, and G. Wetzstein, “Metasdf: Meta-learning signed distance functions,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 10 136–10 147, 2020.

[24] E. R. Chan, K. Nagano, M. A. Chan, *et al.*, “Generative novel view synthesis with 3d-aware diffusion models,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 4217–4229.

[25] Y. Chen, S. Liu, and X. Wang, “Learning continuous image representation with local implicit image function,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 8628–8638.

[26] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[27] Y. Shi, J. Huang, H. Zhang, X. Xu, S. Rusinkiewicz, and K. Xu, “Symmetrynet: Learning to predict reflectional and rotational symmetries of 3d shapes from single-view rgb-d images,” *ACM Transactions on Graphics (TOG)*, vol. 39, no. 6, pp. 1–14, 2020.

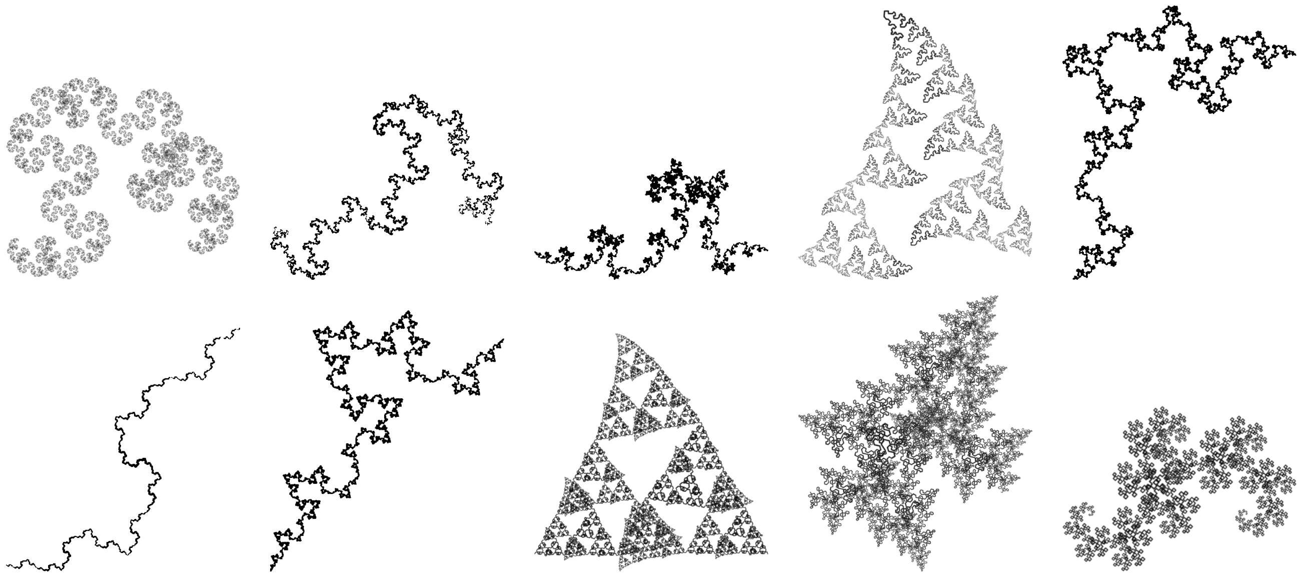

Figure 5. Additional example fractals generated with the method outlined in this paper.

A. Additional Examples

See Figure 5 for additional examples of fractals generated by the approach suggested in this paper. All of these fractals were generated with the same symmetry pattern as the Koch curve, but with the unit square as target image.

B. Sierpiński Carpet Symmetry Pattern

For both the Sierpiński and the apple examples in Figure 2, the symmetry patten used to train the model was as follows. $p_0, p_1, \ldots p_{15}$ are the tunable parameters of the model. As a reminder, the quadruplet $(\{s,t\},\{u,v\})$ means that the model includes a linear transformation that maps the line segment $\{s,t\}$ into the segment {u,v}.

$$\begin{aligned} L = \{ & (\{p_{1,1}, p_{4,4}\}, \{p_{1,1}, p_{2,2}\}), (\{p_{1,1}, p_{4,4}\}, \{p_{1,2}, p_{2,3}\}), (\{p_{1,1}, p_{4,4}\}, \{p_{1,3}, p_{2,4}\}), (\{p_{1,1}, p_{4,4}\}, \{p_{2,1}, p_{3,2}\}), \\ & (\{p_{1,1}, p_{4,4}\}, \{p_{2,3}, p_{3,4}\}), (\{p_{1,1}, p_{4,4}\}, \{p_{3,1}, p_{4,2}\}), (\{p_{1,1}, p_{4,4}\}, \{p_{3,2}, p_{4,3}\}), (\{p_{1,1}, p_{4,4}\}, \{p_{3,3}, p_{4,4}\})\} \end{aligned}$$

This symmetry pattern is illustrated in Figure 6. As with the Koch example, the $p_{i,j}$ are initialized in arbitrary positions and optimized so that the SDF of the generated figure matches the SDF of the target image.

C. Learning Symmetry Patterns

The symmetry patterns discussed in Sections 2 and B are all pre-specified. That is, the symmetry pattern is known ahead of time and the optimization occurs over the location of the control points $\mathcal{P}$. I also consider the problem of learning the symmetry pattern. This can be done by replacing each of the pairs of points in the symmetry pattern with a weighted combination of all of the points. Each element l_i of L is then:

$$l_i = \{f_{1,i}(\mathcal{P}), f_{2,i}(\mathcal{P})\}, \{f_{3,i}(\mathcal{P}), f_{4,i}(\mathcal{P})\}$$

Where each of the $f_{j,i}$ represent weighted combinations of the points in $\mathcal{P}$, so for example $f_{j,i} = \sum_{p \in \mathcal{P}} w_{j,i,p} p$. Convex combinations could be enforced by requiring the weights for a given output sum to 1: $\sum_{p \in \mathcal{P}} w_{j,i,p} = 1$ for any i, j.

In practice this approach seems to be even more susceptible to getting stuck in local maxima than the approach outlined in the main paper. See Figure 7 for examples.

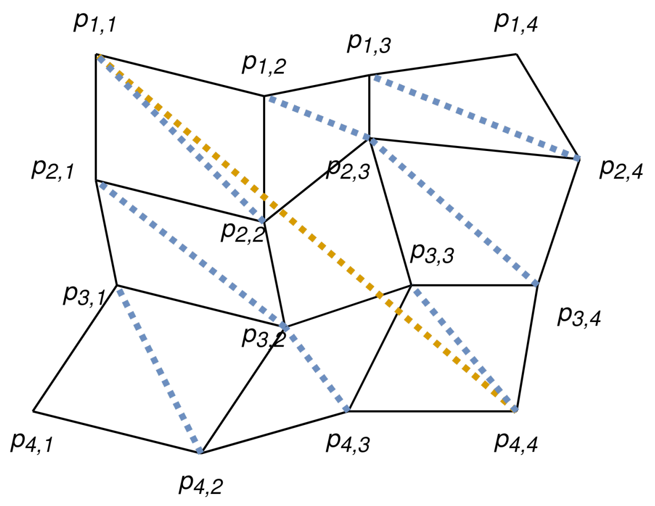

Figure 6. The symmetry pattern used to generate both the Apple and Sierpiński patterns in Figure 2. The orange line is mapped to each of the blue lines via learned linear transformations controlled by the location of the control points $p_{i,j}$.

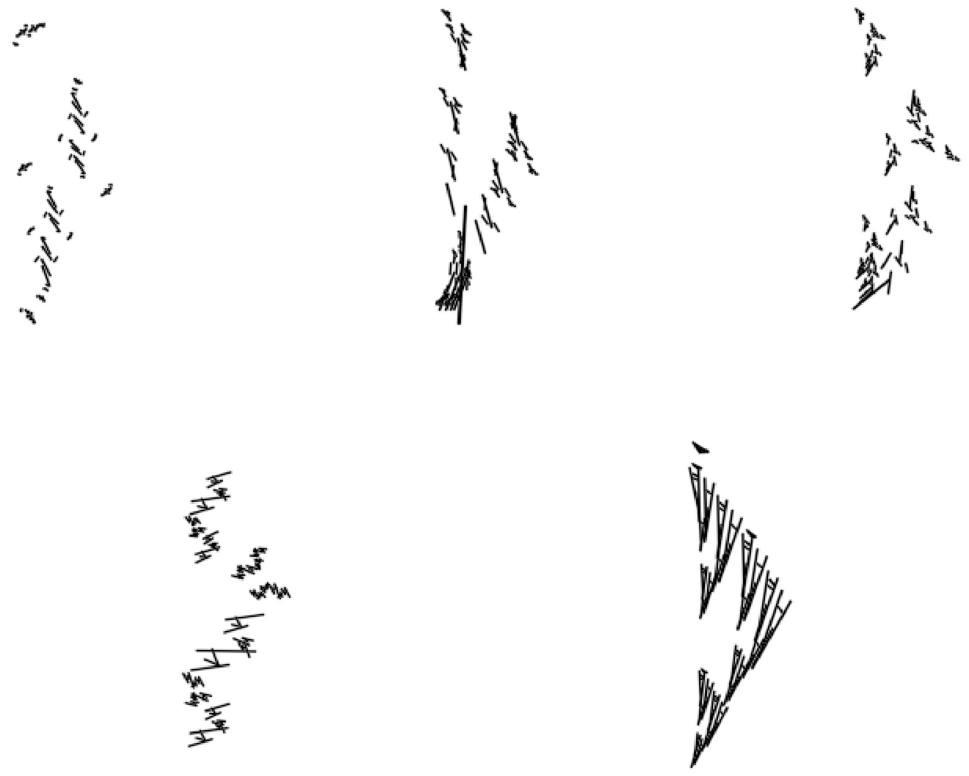

Figure 7. Some examples of local optima encountered when trying to learn the symmetry pattern of the Koch curve.