IFS Visualization Code Documentation

Andrew DeLapo

August 2020

1 Introduction

This document gives some background information for understanding the features of the programs written in this GitHub repository. Primarily, this is a tutorial for using the IFS Visualization program written in Python and which includes a user interface. A version of the program written in Processing is also included in the repository, but the program has no user interface.

This documentation uses the notation and vocabulary from my paper *Bernoulli Randomness and Biased Normality*, available as a pre-print on arXiv here [1]. Additional mathematical background specific to *IFS codes*, as described by Michael Barnsley in [2], is contained in the next section. My paper cites the Processing code in this GitHub repository as the code which generated the images appearing in the paper.

2 IFS Codes

For the definitions of *normal*, *biased normal*, *iterated function system*, the *random iteration algorithm*, and the *determined iteration algorithm*, see [1].

Definition 2.1. An *affine transformation* in $\mathbb{R}^2$ is a function $f : \mathbb{R}^2 \to \mathbb{R}^2$ with associated values $a, b, c, d, e, f \in \mathbb{R}$ such that

$$f\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

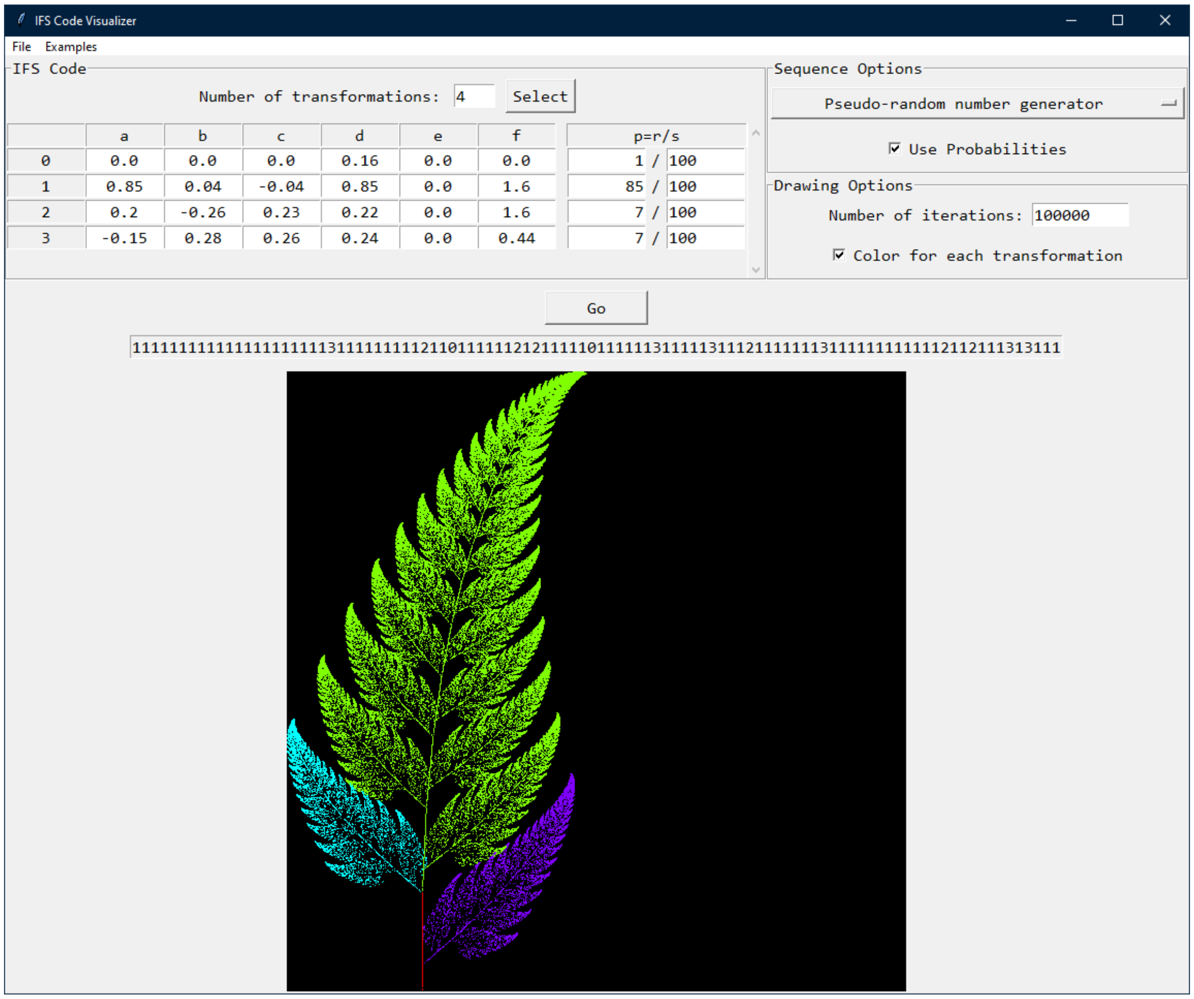
Definition 2.2. Let $\{\mathbb{R}^2; f_0, f_1, \ldots, f_{n-1}; p_0, p_1, \ldots, p_{n-1}\}$ be an iterated function system where each f_i is an affine transformation with associated values $a_i, b_i, c_i, d_i, e_i, f_i \in \mathbb{R}$ as in Definition 2.1. Then the *IFS code* for the iterated function system is the table

i	a	b	c	d	e	f	p
0	a_0	b_0	c_0	d_0	e_0	f_0	p_0
1	a_1	b_1	c_1	d_1	e_1	f_1	p_1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n-1$	a_{n-1}	b_{n-1}	c_{n-1}	d_{n-1}	e_{n-1}	f_{n-1}	p_{n-1}

Example 2.3. The Barnsley fern has the IFS code

i	a	b	c	d	e	f	p
0	0	0	0	0.16	0	0	0.01
1	0.85	0.04	−0.04	0.85	0	1.60	0.85
2	0.20	−0.26	0.23	0.22	0	1.60	0.07
3	−0.15	0.28	0.26	0.24	0	0.44	0.07

These values are how they appear in *Fractals Everywhere* by Michael Barnsley [2].



IFS Code

Number of transformations: 4

	a	b	c	d	e	f	p=r/s
0	0.0	0.0	0.0	0.16	0.0	0.0	1 / 100
1	0.85	0.04	-0.04	0.85	0.0	1.6	85 / 100
2	0.2	-0.26	0.23	0.22	0.0	1.6	7 / 100
3	-0.15	0.28	0.26	0.24	0.0	0.44	7 / 100

Sequence Options

Pseudo-random number generator

☑ Use Probabilities

Drawing Options

Number of iterations: 100000

☑ Color for each transformation

```
1111111111111111111113111111111211011111121211111011111131111131112111111131111111111112112111313111
```

Figure 3.1: IFS Code Visualizer showing the Barnsley fern.

3 IFS Visualizer in Python

The IFS Code Visualizer takes as input an IFS code, a sequence for selecting the transformations, and a number of iterations. The random or determined iteration algorithm is run on the input for the given number of iterations, and the output is shown as a 600x600 pixel image. Additional features include:

- A display of the first 100 digits in the selected sequence
- The ability to export IFS codes as CSV files, which can be imported by the program for later use
- Support for iterated function systems with up to 100 transformations (which require base 100 sequences)
- Support for using Champernowne's sequence or the Copeland-Erdős sequence as input for the algorithm
- Support for applying the biasing algorithm presented in [1]

- The ability to add sequences to the application by writing the sequence as a Python generator

3.1 Specifying an IFS

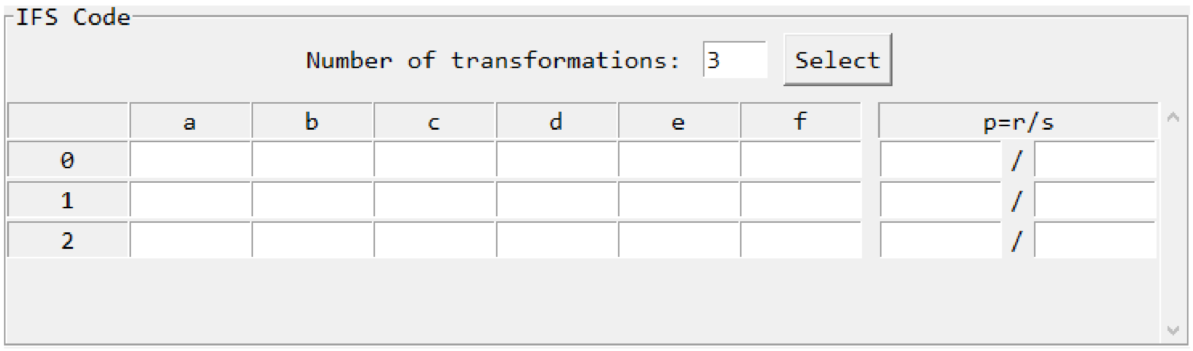
The IFS Code Visualizer uses an IFS code (as in Definition 2.2) to represent an iterated function system with probabilities. Suppose we would like to input the IFS with transformations

$$
\begin{aligned}
f_0(x,y) &= \left(\frac{x}{2}, \frac{y}{2}\right) \\
f_1(x,y) &= \left(\frac{x}{2}, \frac{y+100}{2}\right) \\
f_2(x,y) &= \left(\frac{x+100}{2}, \frac{y+100}{2}\right)
\end{aligned}
$$

and with uniform probabilities $p_0 = p_1 = p_2 = \frac{1}{3}$. One can check that f_0 sends (x,y) to the point halfway between (x,y) and $(0,0)$, f_1 sends (x,y) to the point halfway between (x,y) and $(0,100)$, and f_2 sends (x,y) to the point halfway between (x,y) and $(100,100)$. Then the IFS code for the system is

i	a	b	c	d	e	f	p
0	0.5	0	0	0.5	0	0	1/3
1	0.5	0	0	0.5	0	50	1/3
2	0.5	0	0	0.5	50	0	1/3

To specify this IFS code into the program, first select the number of transformations and click `Select`. The corresponding number of rows will appear in the table below the entry as in Figure 3.2.



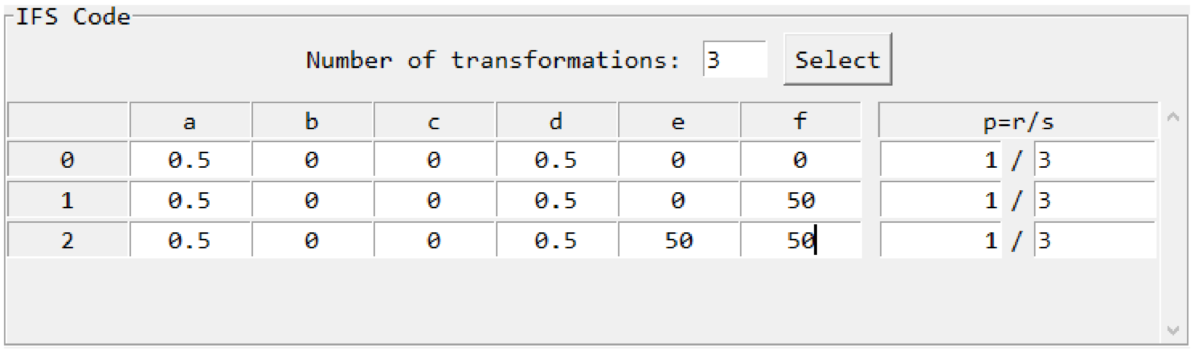
Number of transformations: 3

	a	b	c	d	e	f	p=r/s
0							/
1							/
2							/

Figure 3.2: IFS code entry accepting 3 transformations.

Next, enter the values for each transformation in the table. For entering the probabilities, the entries for the numerator and denominator of each probability are separate. For the current example, the table should be filled as in Figure 3.3.

Alternatively, examples of IFS codes can be loaded into the IFS Code Visualizer by selecting one from the Examples menu. See Section 3.6 for information on adding more examples to this menu.



IFS Code

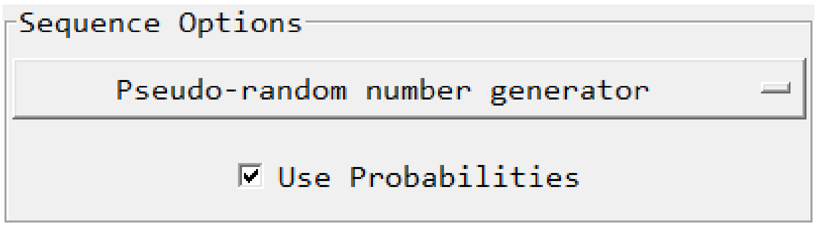
Number of transformations: 3 Select

	a	b	c	d	e	f	p=r/s
0	0.5	0	0	0.5	0	0	1 / 3
1	0.5	0	0	0.5	0	50	1 / 3
2	0.5	0	0	0.5	50	50	1 / 3

Figure 3.3: IFS code entry with the 3 transformations entered.

3.2 Selecting a Sequence

The next step is to specify how the transformations are selected at each step in iteration. The IFS Code Visualizer has three built-in options. These options are found in the drop-down menu of the `Sequence Options` section of the program.



Sequence Options

Pseudo-random number generator

☑ Use Probabilities

Figure 3.4: IFS code entry with the 3 transformations entered.

The `Pseudo-random number generator` option uses Python's built-in random number generator, and the random iteration algorithm is applied. If the `Use Probabilities` button is checked, then at each iteration, the transformation f_i is chosen with probability p_i. Otherwise, each transformation is chosen with uniform probability; if there are n transformations, then f_i is chosen with probability $\frac{1}{n}$.

If any other sequence is selected, then the determined iteration algorithm is used, and the transformations are selected according to the digits in the sequence. The IFS Code Visualizer has Champernowne's constant and the Copeland-Erdős constant built-in. Let n be the number of transformations in the IFS, and let $s_0, s_1, \dots, s_{n-1}$ be the denominators of the probabilities which are specified in the IFS code.

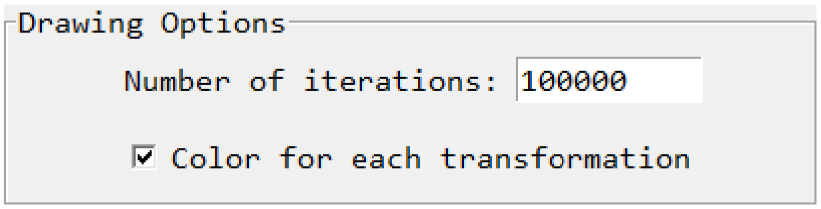
- If `Use Bias` is selected, then the biasing algorithm presented in [1] is applied. This means the selected sequence will be assumed to be a normal sequence in base $b = \text{lcm}(s_0, s_1, \dots, s_{n-1})$, as required by the biasing algorithm. If $s_0, s_1, \dots, s_{n-1}$ are such that $b > 100$, then an error will occur, as sequences in bases greater than 100 are not currently supported. The biasing algorithm will then output a base n biased normal sequence, and this sequence will be used in the determined iteration algorithm.

- If `Use Bias` is not selected, then the selected sequence will be assumed to be a base n sequence, and this sequence will be used in the determined iteration algorithm.

See Section 3.7 on how to edit the `example_generators.py` file to add more sequence options.

3.3 Colors and Number of Iterations

The number of iterations for the random or determined iteration algorithm can be set. By default, the `Color for each transformation` option is enabled, which means that each transformation is associated with a color, and the dot $(x_k, y_k) = f_i(x_{k-1}, y_{k-1})$ is given the color associated with f_i. The colors are assigned so that the HSV (hue, saturation, value) color for f_i is $(\frac{360i}{n}, 1, 1)$. If the `Color for each transformation` option is disabled, then all transformations are given the same color.


Drawing Options

Number of iterations: 100000

[✓] Color for each transformation

Figure 3.5: Drawing options set with 100,000 iterations and different colors assigned to each transformation.

3.4 Viewing the Attractor of the IFS

After specifying the IFS code, selecting a sequence, and setting the number of iterations, press `Go` to run the random or determined iteration algorithm on the given input. Depending on the number of iterations and computational complexity of the selected sequence, it may take several seconds for the attractor to appear. Figure 3.6 shows the program once it has finished 200,000 iterations of the determined iteration algorithm with Champernowne's sequence on the IFS code from Section 3.1, but with adjusted probabilities so that the biasing algorithm from [1] is applied to the sequence. Notice that the first 100 digits of the sequence are shown above the output image.

3.5 Saving the Attractor of the IFS

The content of the output image can be saved as a PNG file by using the File menu and clicking Save Image. A menu will appear to ask for the filename and location for the saved image.

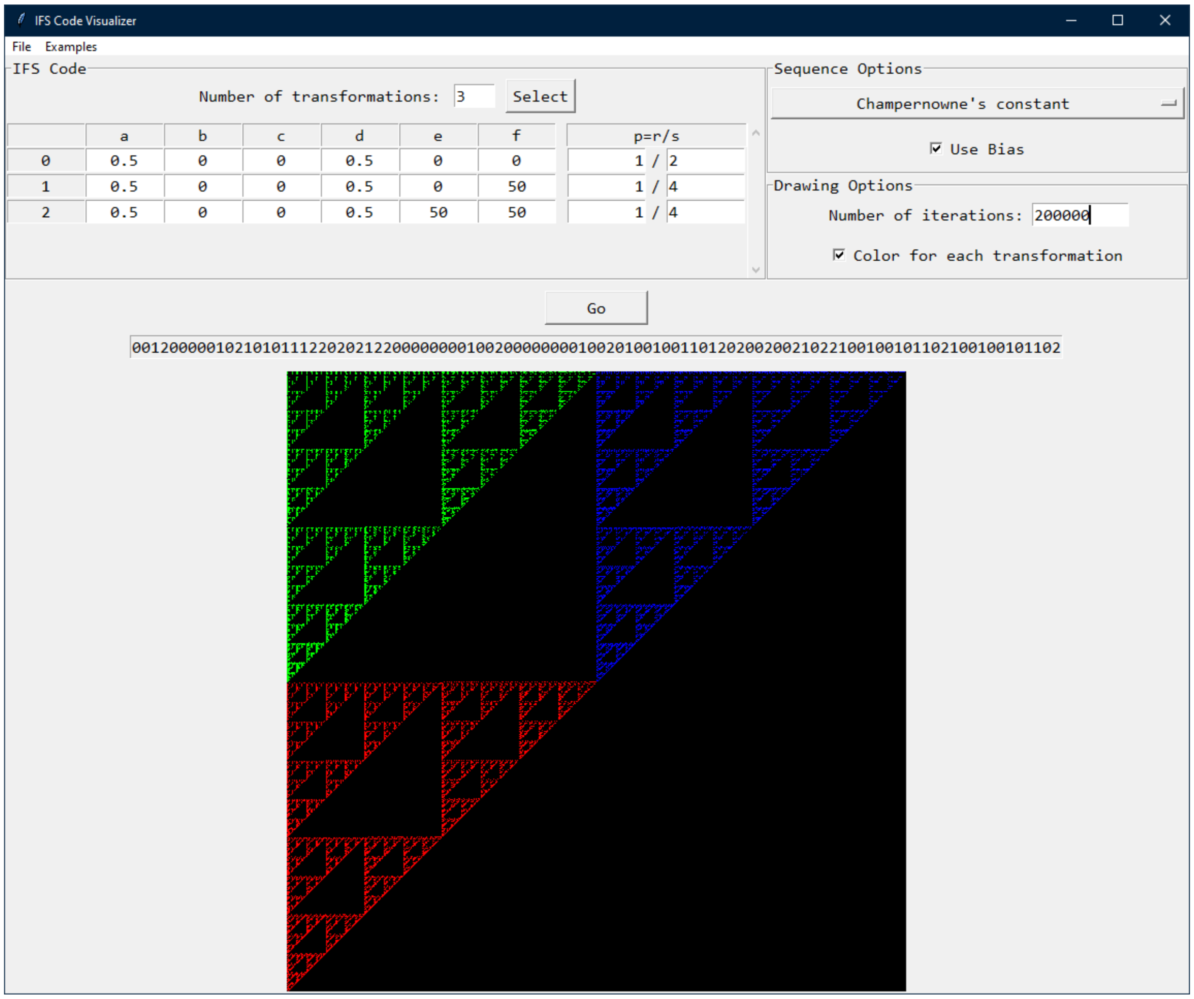
3.6 Exporting the IFS Code

The IFS code examples in the Examples menu are loaded from CSV files in the `ifs_examples` folder located in the same directory as the IFS Code Visualizer. IFS code examples can be added to this menu by exporting IFS codes from the IFS Code Visualizer.

Once an IFS Code is entered, use the File menu and click Export IFS Code. A menu will appear asking for the filename and location for the exported code. To be used in the IFS Code Visualizer, the codes must be exported to inside the `ifs_examples` folder. Once the IFS code is exported, the example can be found in the Examples menu, and the filename is used as the name that appears in the menu. The Examples menu is populated whenever the program starts and whenever an IFS code is exported.

3.7 Adding More Sequence Options (Advanced)

Currently, the only way to add more sequence options to the IFS Code Visualizer is to add the sequence as a Python generator in the `example_generators.py` file. In this section, we will docu-



IFS Code

Number of transformations: 3

	a	b	c	d	e	f	p=r/s
0	0.5	0	0	0.5	0	0	1 / 2
1	0.5	0	0	0.5	0	50	1 / 4
2	0.5	0	0	0.5	50	50	1 / 4

Sequence Options

Champernowne's constant

☑ Use Bias

Drawing Options

Number of iterations: 200000

☑ Color for each transformation

```
0012000001021010111220202122000000001002000000001002010010011012020020021022100100101102100100101102
```

Figure 3.6: IFS Code from Section 3.1, but with probabilities $p_0 = \frac{1}{2}$ and $p_1 = p_2 = \frac{1}{4}$ to demonstrate support for the biasing algorithm from [1].

ment some of the functions appearing in `example_generators.py` and cover an example of adding a new sequence to the examples list.

3.7.1 Important Functions in `example_generators.py`

- `base_repr(number, base=2, padding=0)` is copied from the Python library numpy. It gives the base `base` representation of `number`. This implementation is adapted to support bases up to 100.

- `int(x, base=10)` extends the functionality of Python's built-in `int` function to work for bases up to 100. It converts a string `x` interpreted as a base `base` number to a base 10 integer.

- `concat_generator(generator)` takes a generator as input. Since `generator` might yield strings of length greater than 1 at each iteration, `concat_generator` yields the digits from the generator one digit at a time. This function greatly simplifies the process for creating other generators and is used by `get_champernowne` and `get_copeland_erdos`.

3.7.2 An Example

Besicovitch proved in [3] that the sequence obtained by concatenating the square numbers in base 10 is normal to base 10. The first few digits of the sequence are

$$0.1491625364964811001211441 69\ldots$$

Later results proved that in base b, the sequence obtained by concatenating the square numbers in base b is normal to base b. We will document how this sequence is added to the IFS Code Visualizer. We will write a function `get_besicovitch(base)` which takes as input a base and returns a new generator which yields digits of Besicovitch's sequence with the given base. Note that all generators added to the IFS Code Visualizer must take exactly one positional argument, which is assumed to be the base.

Within `get_besicovitch(base)`, write a generator `square_generator()` which yields the base `base` representation of each square number in a loop. Then have `get_besicovitch(base)` return `concat_generator(square_generator())`. The code could be written as

```
def get_besicovitch(base):
    def square_generator():
        counter = 1
        while True:
            yield base_repr(counter * counter, base=base)
            counter += 1
    return concat_generator(square_generator())
```

Next, `get_besicovitch` needs to be added to the list `all_generators` at the bottom of the file. The elements of the list are tuples, where the first item is the name as it will show in the drop down menu, and the second item is the function which returns the generator. Thus the `all_generators` list should have `(‘‘Besicovitch’s sequence’’, get_besicovitch)` as an element, and the list should appear as

```
all_generators = [("Champernowne's constant", get_champernowne),
                  ("Copeland-Erdos constant", get_copeland_erdos),
                  ("Besicovitch's sequence", get_besicovitch)]
```

Save the changes, and then run `ifs_python_gui.py` to see that the sequence has been added to the list of sequences.

References

[1] Andrew DeLapo. Bernoulli randomness and biased normality. https://arxiv.org/abs/2007.01854, 2020.

[2] Michael Barnsley. *Fractals Everywhere.* Academic Press, Inc., 1988.

[3] A. S. Besicovitch. The asymptotic distribution of the numerals in the decimal representation of the squares of the natural numbers. *Mathematische Zeitschrift*, 39(1):146–156, December 1935.