International Baccalaureate Diploma programme

Higher Level Mathematics Internal Assessment

Investigating how to generate a Koch snowflake fractal, and how this can be used to find a general form of the fractal

2021

Introduction

The rationale behind this exploration is the fascinating fractal patterns of Traditional Batik Shirts. Due to advancements in technology, software such as i-batik can generate software to recursively generate complicated fractals from a simple algorithm.

Fig.1: Display of Batik fabric with a fractal pattern (Tian et al., 2019)

This paper focuses on the generation of the fractal called a Koch snowflake, as well as using the way it is generated to find a general form for a Koch snowflake. As such, How do we generate such a fractal?

Step 1: We are going to explain what a fractal is, as well as what a Koch snowflake is.

Step 2: We use Vectors to "illustrate" the fractal shape. To form Koch snowflake, we need a starting shape termed the "*initiator*" and the way each of the sides changes is called the "*generator*". The number of times we apply the "*generator*" is called a "**generation**", with generation 0 being the "*initiator*". (As we have not yet applied the "*generator*" at **generation 0**) We use Vector transformations and Vector coordinates to change the *initiator* into the generator.

Step 3: To see how the *initiator* transforms using the *generator* , we can use Modular Arithmetic to predict how the fractal looks like by finding a general formula to get all the vector coordinates for any **generation g.**

Step 4: As Step 3 is not truly recursive, thus we let every 2 points in the fractal, to create x additional points based on its "*generator*" . To relate the 2 points to the x number of points, we can convert the points to a complex plane, we apply $e^{i\theta}$ as a rotational transformation, thus recursively relating the points. **Step 4.2:** However, as we are using Python code and Vectors

(to generate the fractal), it is more efficient to use matrices. As such instead of using $e^{i\theta}$, we can use a matrix linear transform.

Step 5 (Conclusion) : Putting all the steps together to show how to recursively generate these functions, as well as formulating a claim for the generalisation of a Koch snowflake.

1. Background information about Fractals

Benoit Mandelbrot coined the term Fractals from the Latin adjective “Fractus” with connotations of “fragmented” and “irregular” – An irregular fragment. Thus, Mandelbrot states that "A fractal is by definition a set for which the Fractal (Hausdorff-Besicovitch) dimension *D* strictly exceeds the topological (Euclidean) dimension."

1.1 Self-similarity

Fig.2:Coastline of Britain, with D ≈ 1.25 (GADM, n.d.)

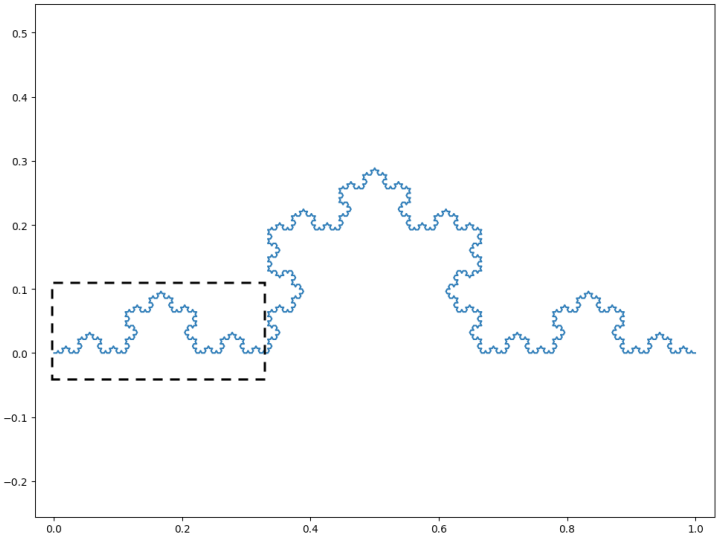


Fig.3:Koch Curve, with D ≈ 1.26 (Created in Python 3.9.2)

As such a fractal need not be self-similar. Self-similarity is like the Koch curve in Fig.3, where the boxed part is a scaled down version of the entire Koch curve. This could repeat infinitely, as we keep scaling to get back itself. On the other hand, the British coast in Fig.2 has many, many cracks and branches as we keep zooming in, or scaling up the image. What connects these 2 images?

1.2 Fractal dimension

Both fig.2 & 3 have non-integer fractal dimension *D*. To make it less abstract, think of *D* to be the degree of space-filling. A curve with *D* very close to 1.0 (such as 1.1) behaves much like an ordinary one-dimensional line, but a curve with D very close to 2.0 (such as 1.9) has a very convoluted shape, much like a two-dimensional surface.

If we take a step back and look at traditional Euclidean shapes:

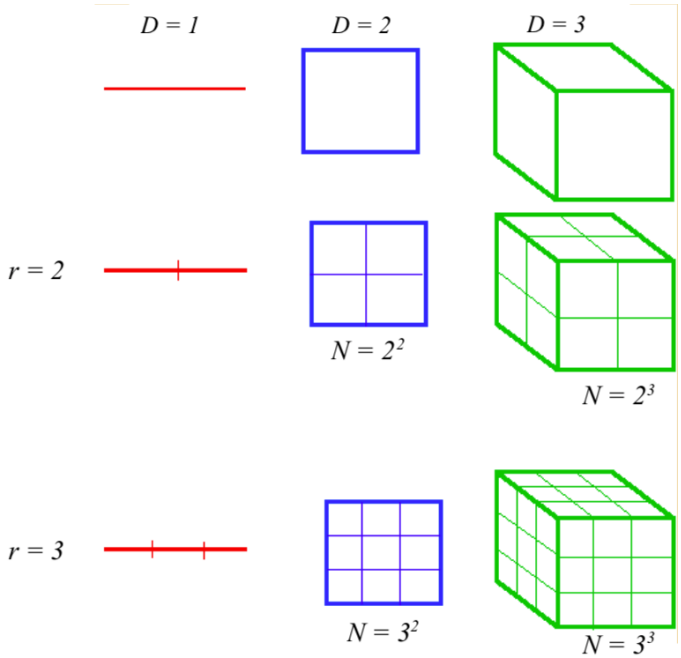


Fig.4:1-dimensional, 2-dimension, 3-dimensional shapes that create copies of itself after scaling, (Fractals & the Fractal Dimension, 2019)

When one scales an object by a factor of *r*, the amount of space it fills up N (one can think of it as the space it can fill up in the original shape, after it is scaled down), will change according to the dimension.

A 2 D object (in this case a square) gives a power of 2 as it forms r^2 copies of the original. When the square is scaled down by a factor of 2, it can fill up 4 times of the original.

When the square is scaled down by a factor of 2, it can fill up 9 times of the original.

Looking at this pattern, we can generalise a formula to find *D*:

$$N = r^{D}$$

$$D = log(N) / log\ (r)$$

2. Generation of a Koch snowflake

One of the most intuitive examples of a fractal is the Koch snowflake, created by Swedish mathematician Helge von Koch, 1904. It is a self-similar fractal as if we were to scale up each side, it would look like the original pattern. More specifically, it scales by the factor of $r = 3$ and when it is scaled down, it can fill up 4 copies of the original shape.

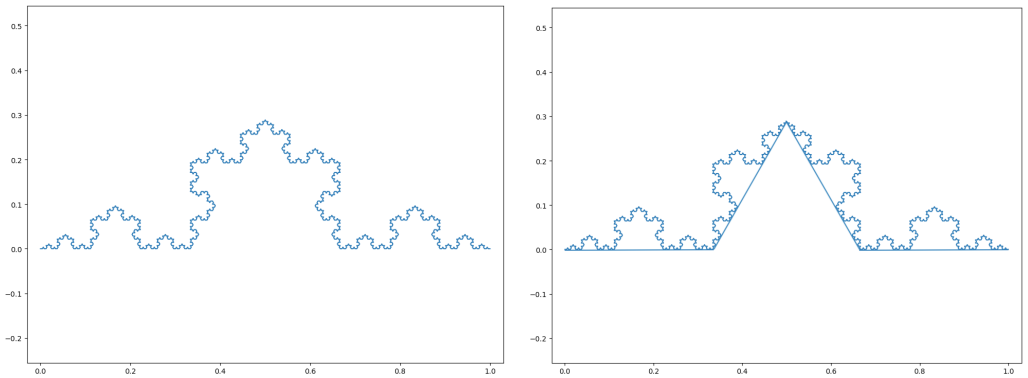


Fig.5:Koch curve on the left and Koch curve shown to be 4 copies on the right (Created in Python 3.9.2)

As such it's fractal dimension is: $D = log\ (4) / log\ (3) \approx 1.26\ (3\ s.f.\ )$

To find a general formula for not only Koch snowflakes with different *initiators*, but also with different generator functions, let us take a look at a Koch snowflake (3 sided Koch curve) :

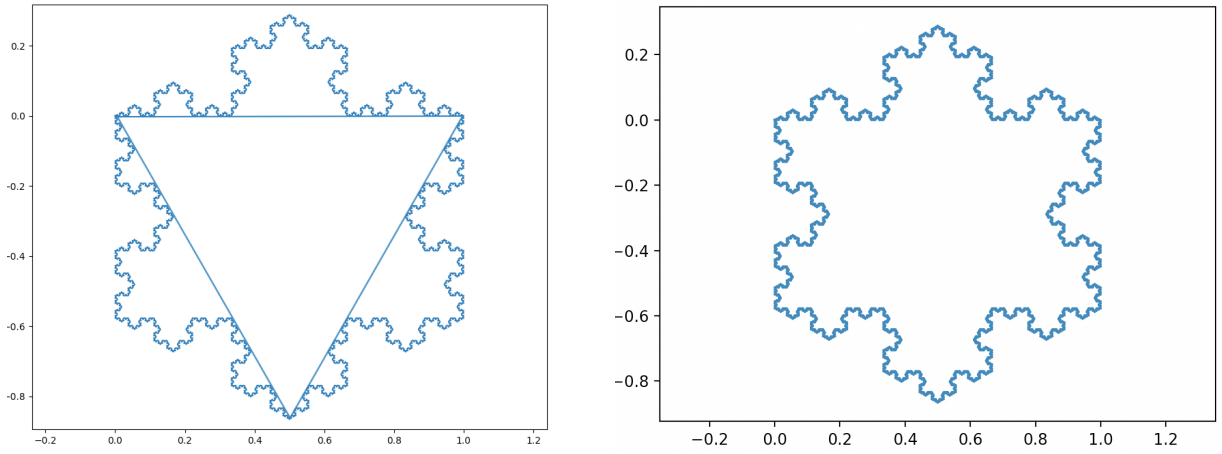
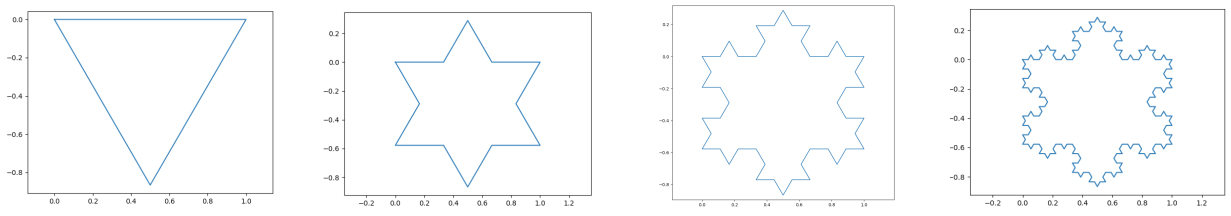


Fig.6:Koch Snowflake shown to be made up of 3 Koch curves (Created in Python 3.9.2)



Generation 0 *Generation 1* *Generation 2* *Generation 3*

Fig.7:Increasing generations of Koch Snowflake using generator function g ={0,-1,-2,0} (Created in Python 3.9.2)

2.1 Formation of Koch snowflake

The *initiator* is the shape that we start with. The *generator* is how each side of the initiator changes:

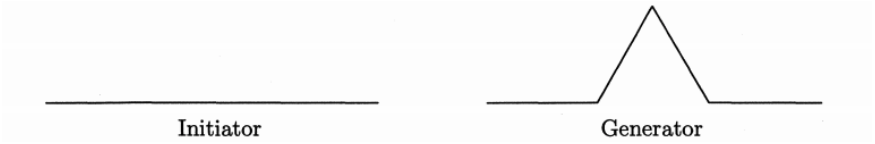


Fig.8:How the initiator changes due to generator function g ={0,-1,-2,0}

Each side of the Koch snowflake is replaced by the generator function:

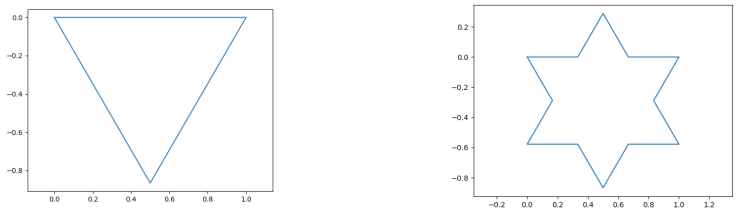


Fig.9:How the Koch Snowflake changes due to generator function g ={0,-1,-2,0}
(Created in Python 3.9.2)

2.2 General illustration of Koch snowflake

To represent the Koch snowflake, we can use the help of vector coordinates:

Vectors, having a line and a direction, allows us to plot coordinates that allow for linear transformations. For simplicity, we can start the graph of the Koch Snowflake at the point of origin (0, 0). How do the vectors apply to the creation of this fractal? We can “draw” the fractal using vectors in a clockwise direction, until we get a closed vector shape:

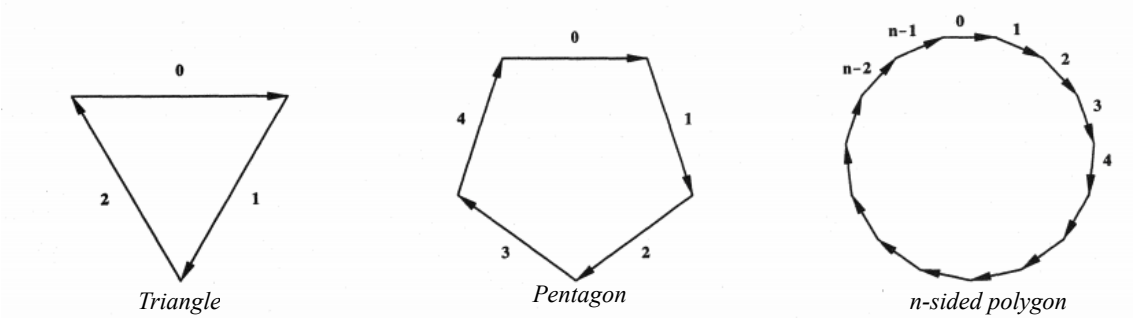


Fig.10:Closed vector polygons, used from (Shakiban & Bergstedt, 2000)

Each side of the shape is labelled from 0 to $n - 1$, where n = *the number of sides of the initiator*. It is important to note that this shape is a regular polygon, with the length of each side being equal, and the angles in the shape are equal.

The easiest way to plot the vectors, is by inscribing the regular polygon into a circle (Fig.11). The radius of the circle is the length of each vector 0,1,2. It is important to note that each vector here is related to the origin (0, 0). From here we can see that we split the circle up into 3 equal parts, meaning that each vector would have the angle, $-2\pi / 3$. (It is "negative" 2π as we are going from vector "0" to "1", to "2", in a clockwise manner).

Just by the translation of the vectors 0,1 and 2 (shown in the extrapolation in Fig.11), each vector is placed from head to tail in a closed vector shape, we get this:

The equilateral triangle initiator that we originally started off with

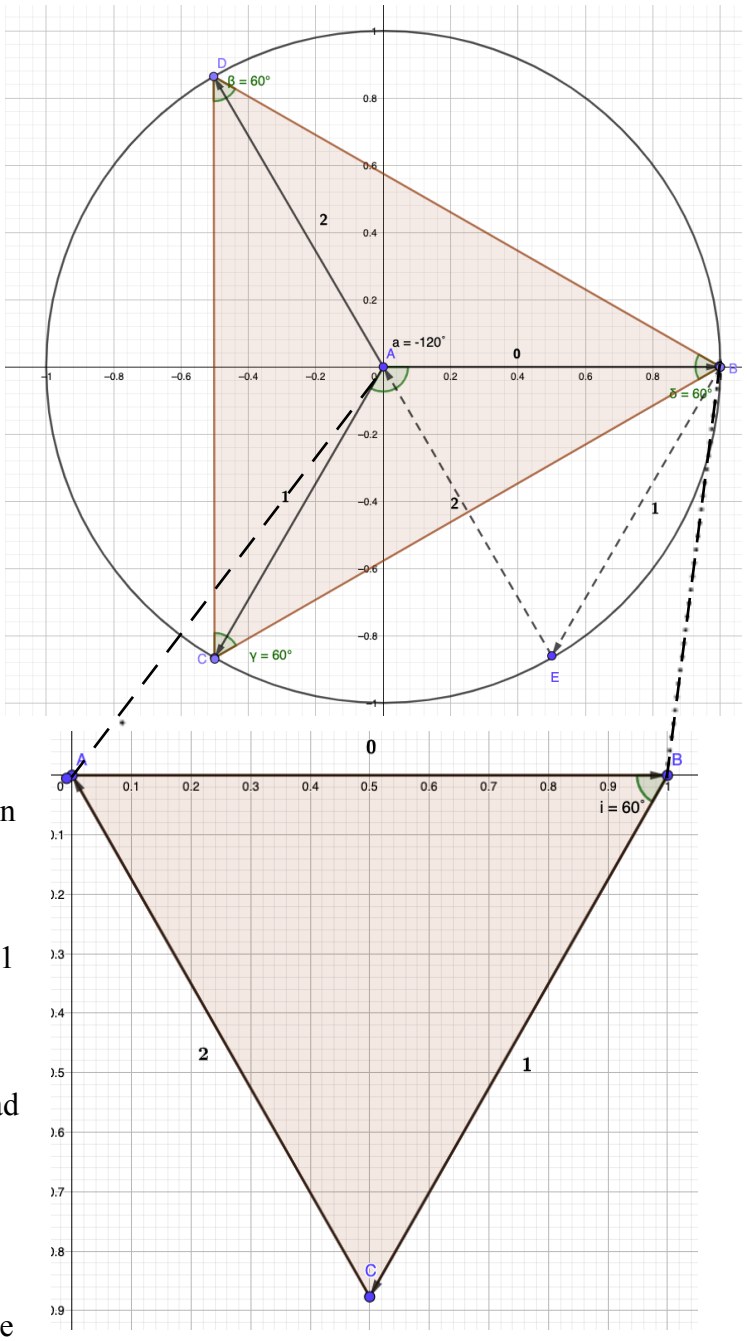


Fig.11: Top:Inscribed regular 3-sided polygon in circle to find vectors 0, 1 & 2 Extrapolated: Translation of vectors to become a closed shape (Created in GeoGebra)

2.3 Generalisation of Koch snowflake using Vectors

To generalise, let us call each vector that we create v and each side of the *initiator* to be *n*.

To create each Koch snowflake, we can use the vectors 0, 1, 2, … , $(n - 1)$. As well as their reflection (π rotation) which is - vectors -0, – 1, – 2, … , – $(n - 1)$.

2.4 Finding the coordinates of each Vector

The coordinate for each vector is C_V , For negative vectors, they follow a π rotation, thus:

$$C_{-V} = - C_V$$

Mentioned earlier, the angle of each coordinate (in the triangle) is a multiple of - 2π/3, depending on which is the vector, gets the formula:

$$\theta_V = - 2\pi (V / n)$$

It is divided by *n* as the angle of each vector splits the circle (2π), into equal portions depending on the number of sides of the initiator *n*.

Finally, each of the vectors are placed originally on a circle, meaning that their coordinates: $C_V = (r \cos \theta_V , r \sin \theta_V)$, where r is 1/scaling factor. This is because r is to compensate for the radius of the circle being the hypotenuse of the triangle that the vector forms. Depending on the scaling factor, the length of the radius will change. As such, this is compensated by dividing the coordinates by *1 / radius length = 1 / scaling factor = r* .

For the *initiator* above, the coordinates are:

Vector	$\theta_v = - 2\pi (V / n)$	$C_v = (r \cos \theta_v , r \sin \theta_v)$
0	$\theta_0 = 0$	$C_0 = (\cos 0 , \sin 0)$
1	$\theta_1 = - 2/3\ \pi$	$C_1 = (\cos -2/3\ \pi , \sin -2/3\ \pi)$
2	$\theta_2 = - 4/3\ \pi$	$C_2 = (\cos - 4/3\ \pi , \sin - 4/3\ \pi)$
– 0	–	$C\text{-}_0 = (- \cos 0 , - \sin 0)$
– 1	–	$C\text{-}_1 = (\cos 2/3\ \pi , \sin 2/3\ \pi)$
– 2	–	$C\text{-}_2 = (\cos 4/3\ \pi , \sin 4/3\ \pi)$

Table 1: List of all Coordinates of Vectors 0, -0,1,-1,2,-2

Plotting the vectors we get Fig.12:

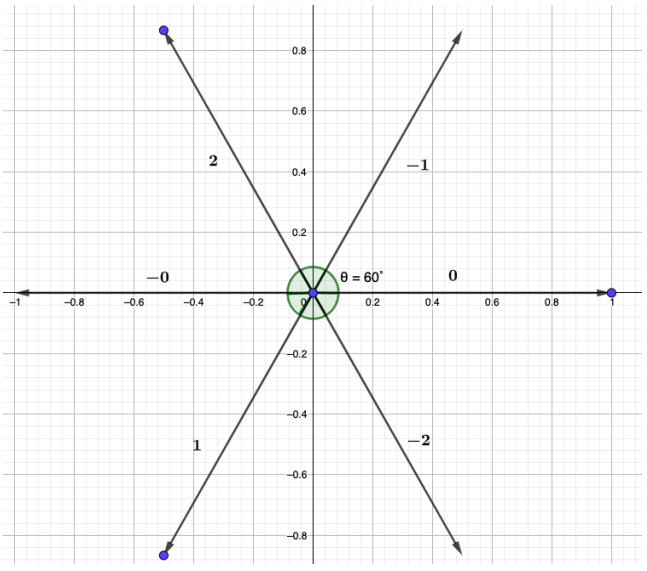


Fig.12: Visual representation of all the vectors 0,-0,1,-1,2,-2 with unit length (Created in GeoGebra)

3. Properties of a Generator function

For a generalised Koch snowflake, we realise that we only can use the vectors and the negative vectors created in each initiator (From Fig8). This is seen in the Koch snowflake:

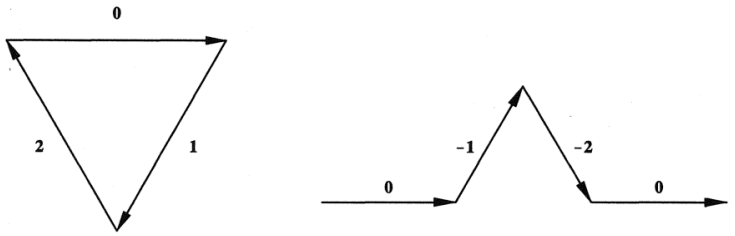


Fig.8.1: Fig.8 represented using vectors, from (Shakiban & Bergstedt, 2000)

In this case the generator function is : g = { 0 , -1 , -2 , 0 }, *g* indicates how each element in the generator changes the side, in Fig.14 it changes the vector side 0. As such, it changes vector side 0 to 4 new vectors of 0, –1, –2 and 0. Now, how can we apply this to the other sides of vectors 1 and 2?

3.1 Using Modular arithmetic

We can think of Modular arithmetic like an everyday clock, when it is **14 o'clock**, we know that it is **2 pm**. The clock is in 'base' 12, as it repeats itself after every 12 numbers. The number 14 refers to the 2nd number after the repeat, the number 2. We can also think of it as 14/12 = 1 remainder 2, where the remainder is what we are interested in. Conventionally, we can say that 14 mod 12 = 2 , where it follows *x mod z = V* , where x is the dividend, z is the divisor and V is the remainder.

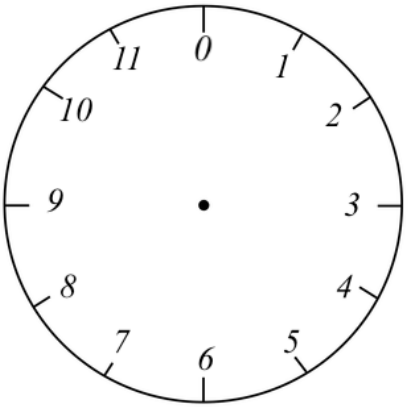
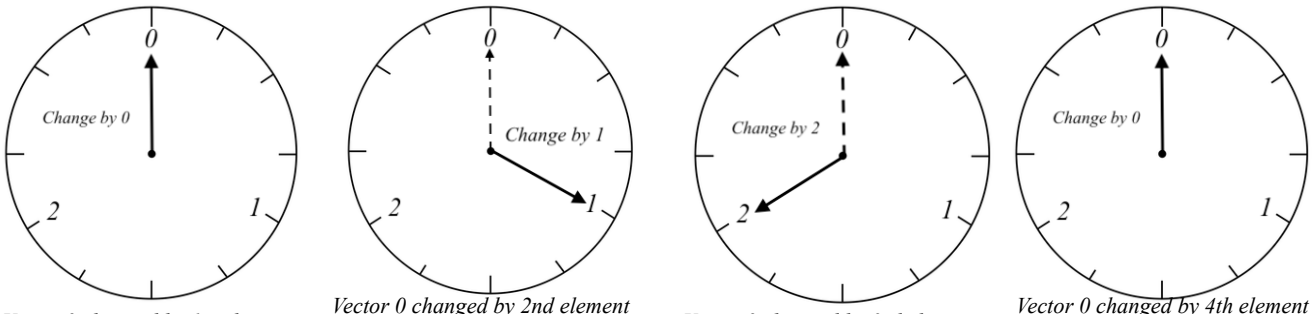


Fig.13: 12 numbered clock

With this, we can think of the *initiator* like a 3 numbered clock. For g = { 0 , -1 , -2 , 0 }

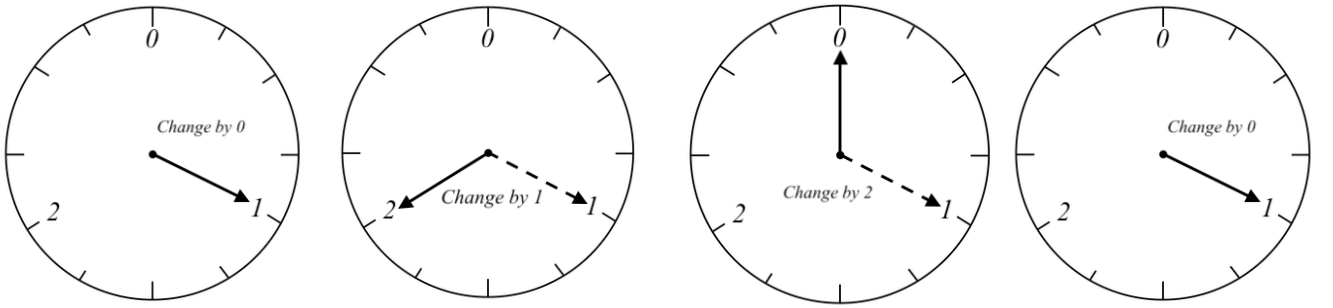
Disregarding the negative sign, Vector 0 becomes: vector 0, 1, 2, 0



Vector 0 changed by 1st element *Vector 0 changed by 2nd element* *Vector 0 changed by 3rd element* *Vector 0 changed by 4th element*

Fig.14: How each element in g changes the vector side 0

Disregarding the negative sign, Vector 1 becomes: vector 1, 2, 0, 1



Vector 1 changed by 1st element *Vector 1 changed by 2nd element* *Vector 1 changed by 3rd element* *Vector 1 changed by 4th element*

Fig.15: How each element in g changes the vector side 1

Therefore we get the formula: *(|V| + |g|)mod n) = corresponding changed vector*

Where, n = number of sides of the *initiator*, V is the vector and g is the generator function

We can **disregarded the negative sign** here as it is not indicating a negative number but indicating a π rotation

3.2 General formula to apply generator to each side

Finally, putting it all together, we can get the formula for how to generate the corresponding vectors when one has a generator. We also must not forget to compensate for **"disregarding the negative sign"** :

(sign of V • sign of g) • ((|V| + |g|) mod n) = corresponding changed vector

Once we have this formula, we can generate the different vectors for each generation:

Generation	0	1	2	0	1	2	0	1	2
Vector	0	0	0	1	1	1	2	2	2
			–1			– 2			– 0
			– 2			– 0			–1
			0			1			2
		–1	–1		– 2	–1		– 0	– 0
			2			2			1
			0			0			2
			–1			–1			– 0
		– 2	– 2		– 0	– 0		– 1	–1
			0			1			2
			1			2			0
			– 2			– 0			–1
		0	0		1	1		2	2
			–1			– 2			– 0
			– 2			– 0			–1
			0			1			2

Table 2: Generated List of all vectors needed from generation 0 to 2, from the formula above

Plotting out all vector coordinates, we get (Fig.16), Koch snowflake for the 2nd generation:

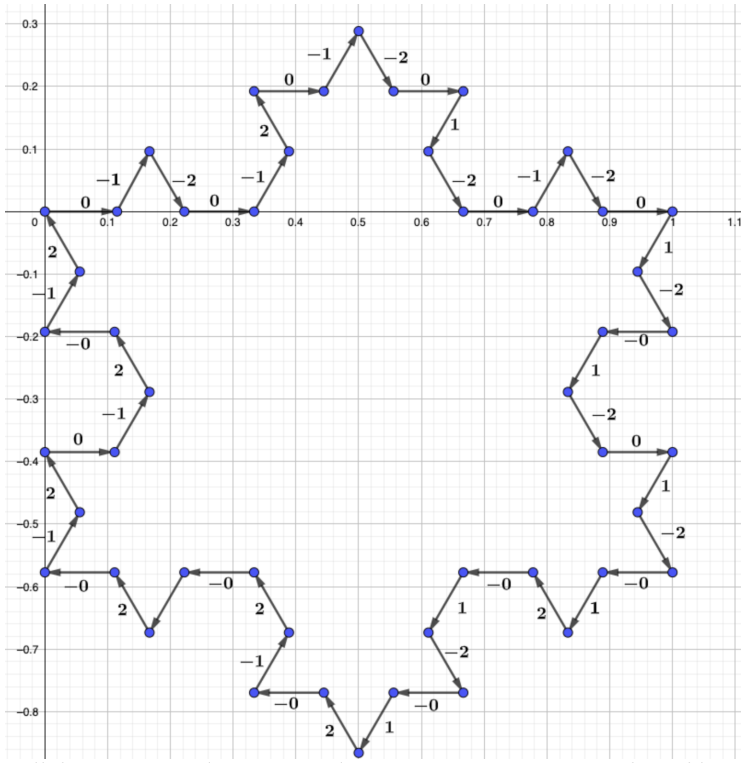


Fig.16: Adding all the vector coordinates according to generation 2 generated in table 2 (Created with GeoGebra)

However, for increasing generations, this method becomes very slow and tedious:

Each of the length of the line will change based on r (1/scaling factor), in this case it is $1/3^{gen}$, where gen is the generation of the fractal. Also fig.16 is the coordinates of each vector relative to (0, 0). Each vector is added up in sequence from fig.12, from their head to tail giving the final output as Fig. 16.

This is because **for every generation w**, we will get **$3 \cdot 4^w$ vectors**. This is because we generate 4 vectors for each vector in the generation, and we have 3 sides total in the triangle. Just for the 5th generation, we would have to add up $3 \cdot 4^5 = 3072$ vectors

Although plausible, it is very difficult to implement this for generator functions and initiators that have more vectors included. Is there a faster way to see how the generator function affects the types of Koch snowflake?

4. Forming the Recursive relationship between coordinates & generator function g

If we can create a recursive relationship between 2 points, this can be applied to the rest of the points in a very efficient and fast manner.

How does it work?

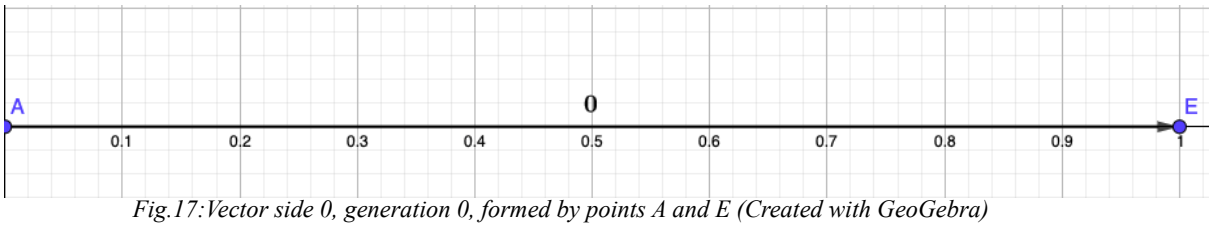


Fig.17:Vector side 0, generation 0, formed by points A and E (Created with GeoGebra)

We have a function that, for every 2 points, (in this case A and E), it will create points based on the generator: { 0 , -1 , -2 , 0 }

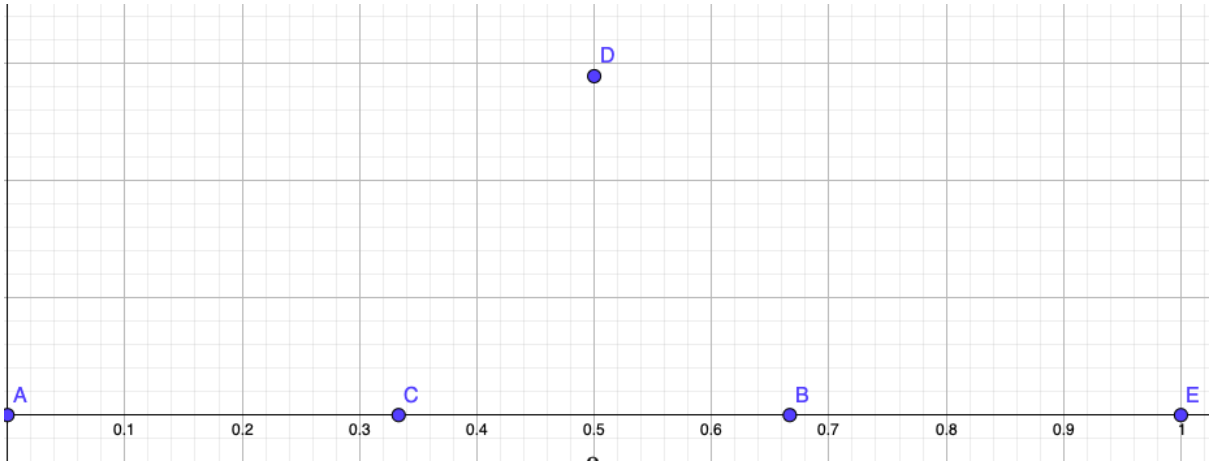


Fig.18:Vector side 0, Generation 1, formed by points A to E (Created with GeoGebra

It creates 3 new points (B, C, D)

<u>As such every 2 points creates 3 more points</u>

We can create a "find" function, to relate the original 2 points to the other 3 points. With that, we can set every 2 points to carry out that "find" function. By doing so, we can have a recursive relationship for all the points in one generation to the next.

<u>Forming a Recursive relation:</u> Point A = Point X_h , **Point B** = Point X_{h+1}, **Point C** = Point X_{h+2}, **Point D** = Point X_{h+3}, **Point E** = Point X_{h+4}

This begs the question, how do we relate the points that are not on the side itself, like point C?

4.1 Using the Complex plane

In Fig. 19, we can see that the relationship between point A and point C is a rotation by π/3, followed by the translation of the vector to point B:

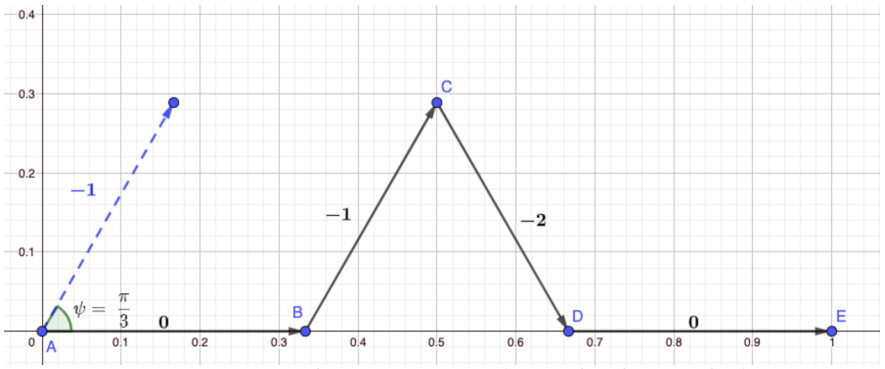


Fig.19: Vector side 0, Generation 1 (Created with GeoGebra)

By converting these vectors to points on a complex plane, we can easily do these rotations. The complex plane is fantastic for doing rotations of this kind as the whole idea of the complex plane is based on the number i , where it is defined as:

$$i^2 = -1$$

Normally, when we rotate a point (**a**, **b**) by π/2, we can see that the point becomes (**– b**, **a**)

The point (3, 2) becomes (–2, 3) after a π/2 rotation

This gives i , a very unique property, where:

$$i \bullet (\boldsymbol{a} + i\,\boldsymbol{b}) = (-\boldsymbol{b} + i\,\boldsymbol{a})$$

If we think of it as coordinates, the point(**a**, **b**) become (**– b**, **a**)

We see from the equation that multiplying i to a complex number, gives a rotation of π/2

That is why we see that the complex plane i is exactly π/2 from the real plane.

Conventionally, we can use $r\,(\cos\theta + i \sin\theta)$ to denote a complex number, it is the same concept as Finding $C_V = (r \cos\theta_V , r \sin\theta_V)$, but instead the "y" coordinate is in terms of i. This is actually the same as the notation $re^{i\theta}$ (complex number in exponential form). However, what does it mean to raise the number e, to an imaginary constant?

In truth, e is actually a function, where mathematicians define it as:

$$e = \lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n$$

Raising e to a power x means that:

$$e^x = \lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^{nx}$$

$$e^x = \lim_{n \to \infty} \left(1 + \frac{x}{nx}\right)^{nx}$$

$$e^x = \lim_{m \to \infty} \left(1 + \frac{x}{m}\right)^{m}, \ where\ m = nx$$

When $x = i\theta$,

$$e^{i\theta} = \lim_{m \to \infty} \left(1 + i\frac{\theta}{m}\right)^{m}$$

Raising a power of a complex number is a rotation + scaling factor as it is a multiplication of 2 complex numbers. As we increase m , the scaling factor becomes closer to 1+ $i0$ and the number of rotations m, increases to infinity. Lastly, the length of each infinitely small piece of $e^{i\theta}$ is θ / m , and there are m pieces. Thus, the coordinate would be $\theta / m \bullet m = \theta$, corresponding to the coordinate $\cos\theta + i \sin\theta$.

Using we also can prove this by using binomial theorem and calculus (MacLaurin series):

$$e^{i\theta} = \lim_{m \to \infty} \left(1 + \frac{i\theta}{m}\right)^{m}$$

$$= \lim_{m \to \infty} \left(\sum_{k=0}^{m} \binom{m}{k} \left(\frac{i\theta}{m}\right)^{m-k}\right)$$

$$= \lim_{m \to \infty} \left(1 + \frac{m}{1!}\left(\frac{i\theta}{m}\right) + \frac{m(m-1)}{2!}\left(\frac{i\theta}{m}\right)^2 + \ldots + \frac{m(m-1)\ldots(m-k+1)}{k!}\left(\frac{i\theta}{m}\right)^k\right)$$

$$= \lim_{m \to \infty} \left(1 + \frac{m}{m}\left(\frac{i\theta}{1!}\right) + \left(\frac{m}{m}\right)\left(\frac{m-1}{m}\right)\left(\frac{(i\theta)^2}{2!}\right) + \ldots + \left(\frac{m}{m}\right)\left(\frac{m-1}{m}\right)\ldots\left(\frac{m-k+1}{m}\right)\left(\frac{(i\theta)^k}{k!}\right)\right)$$

$$= 1 + \frac{i\theta}{1!} + \frac{(i\theta)^2}{2!} + \ldots + \frac{(i\theta)^k}{k!} + \ldots.$$

Expanding the value ($i\theta$), we get the Maclaurin series for $\cos\theta$ and $\sin\theta$:

$$= \left(1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \ldots\right) + i\left(\theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \ldots\right)$$

$$= \cos\theta + i\sin\theta$$

This is because,

$$\lim_{m \to \infty} \left(\frac{m-j}{m} \right) = \lim_{m \to \infty} \left(1 - \frac{j}{m} \right)$$
$$= 1 \ , \ where \ j \in \mathbb{Z}$$

As such, we can finally relate point C. **Where** Point $X_{h+2} = \mathbf{e}^{i(\pi/6)} \bullet$ Point X_{h+1} + Point X_{h+1}

However, it becomes difficult to input this version of the rotational transformation in python code as we are constantly converting back and forth between the native python vector form and the complex number form. In that case, is there a better way to implement this rotation?

4.2. Matrices and linear transformations

Natively in python, it is easier and more efficient to code using matrices. As a result, instead of using a rotational formula, we can instead, use a rotation matrix.

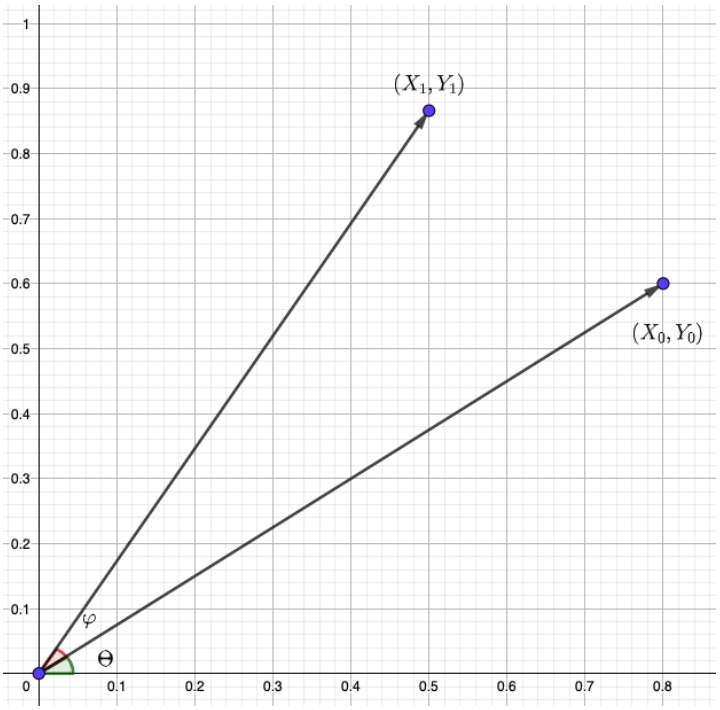


Fig.20:Coordinate vectors with different angles but same length

As all of the vectors are on a circle, they all have the same length (radius of circle), and are related by their angles! This then begs the question, how do we rotate a vector so that it becomes related to another vector? We can use the linear transformation of rotating matrices.

Using the formula for the coordinates that we derived from part 3,

From Fig. 20 , the coordinates are:

$X_0 = r \cos \boldsymbol{\theta}$, $Y_0 = r \sin \boldsymbol{\theta}$: $C_1 = (r \cos \boldsymbol{\theta} , r \sin \boldsymbol{\theta})$

$X_1 = r \cos (\boldsymbol{\theta} + \varphi)$, $Y_1 = r \sin (\boldsymbol{\theta} + \varphi)$: $C_2 = (r \cos (\boldsymbol{\theta} + \varphi) , r \sin (\boldsymbol{\theta} + \varphi))$

Using trigonometric identities:

$X_1 = r \cos(\boldsymbol{\theta} + \varphi)$ $= r(\cos\boldsymbol{\theta}\cos\varphi - \sin\boldsymbol{\theta}\sin\varphi)$ $= (r\cos\boldsymbol{\theta})\cos\varphi - (r\sin\boldsymbol{\theta})\sin\varphi$ $= X_0\cos\varphi - Y_0\sin\varphi$	$Y_1 = r \sin(\boldsymbol{\theta} + \varphi)$ $= r(\sin\boldsymbol{\theta}\cos\varphi + \cos\boldsymbol{\theta}\sin\varphi)$ $= (r\sin\boldsymbol{\theta})\cos\varphi + (r\cos\boldsymbol{\theta})\sin\varphi$ $= Y_0\cos\varphi + X_0\sin\varphi$ $= X_0\sin\varphi + Y_0\cos\varphi$

Writing this in a 2 dimensional matrix form:

$$\begin{bmatrix} X_1 \\ Y_1 \end{bmatrix} = \begin{bmatrix} X_0\cos\varphi - Y_0\sin\varphi \\ X_0\sin\varphi + Y_0\cos\varphi \end{bmatrix}$$

$$= \begin{bmatrix} X_0 \\ Y_0 \end{bmatrix} \bullet \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix}$$

As such, we get the rotational matrix. With this, the vector $[X_0 , Y_0]$ can be transformed to the vector $[X_1 , Y_1]$ when one multiples the rotational matrix.

Relating back to point C, **Point X_{h+2} = rotational matrix for $\pi/3$ • Point X_{h+1} + Point X_{h+1}**

Using python, we let C_V is a coordinate on the initiator:	
Point X_h = [C_V] Point X_{h+4} = [C_V + 1] Point X_{h+1} = (2 • Point X_h + Point X_{h+4}) / 3 Point X_{h+3} = (2 • Point X_{h+4} +Point X_h) / 3 Point X_{h+2} = Point X_{h+1} + ([Point X_{h+1}] • rotational matrix for $\pi/3$)	This means that **for every 2 points**, it will generate an **<u>additional 3 points</u>** (Point X_{h+1}, Point X_{h+3}, Point X_{h+3}) . This is recursive as it repeats for every single 2 points:

5. Conclusion

Now, we have the tools to define what a n-Koch snowflake of Vector = V!

1. *Initiator*
    a. It is a n-sided shape, with V = 0, 1, 2... $n - 1$
    b. It is a regular polygon with $\theta_V = -2\pi (V / n)$
    c. It has vector coordinates $C_V = (r \cos \theta_V , r \sin \theta_V)$
    d. It can use –V with $C_{-V} = - C_V$

2. *Generator function g*
    a. g = { 0, u, 0)
        i. where there is u number of V and -V, in the form: V or -V, ….,
    b. It only uses V from the *initiator* and the –V of the *initiator*
    c. Like the *initiator*, the g must form a closed vector shape

3. Generating the function
    a. How each element in g changes each vector is by:
    *(**sign of V • sign of g**) • ((|V| + |g|) mod n) = corresponding V*
    b. When using python code we use the rotation matrix
    $$\begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix}$$
    to form relationships between any 2 vectors
    c. Or else we can use $e^{i\theta} \cdot C_V$ after converting to the complex plane
    d. This allows us to get a recursive relationship

Lastly, we go back to the idea of Fractal dimension, and to see how much space the fractal fills, we can use the formula from Section 2: $D = \log(N) / \log (r)$

Where, the **number of copies** (**N**) formed is determined by the number of elements in the generator g and **r** is the **scaling factor**, in this case, the length of each line

(eg. if we have 3 elements in g, it creates 3 more vectors, forming 3 more copies of itself)

Let us generate beautiful Koch snowflake patterns, with all the python codes used are placed in the appendix.

5.1 Using other generators for the equilateral triangle *initiator*

Fig. 21turns into Fig. 22 (for g = { 0, -1, -2, 0})

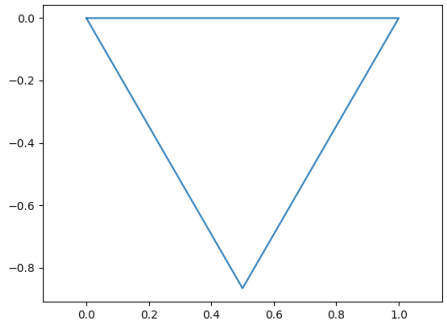


Fig. 21: Triangle initiator (Created with Python 3.9.2)

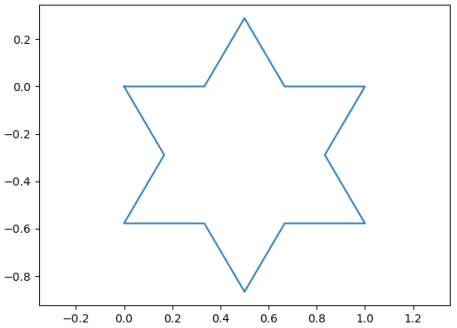


Fig. 22: Generation 1 (Created with Python 3.9.2)

If we change the generator to {0 , -2, -1, 0}, we get: $D = log\ (4)\ /\ log\ (3) \approx 1.26$ (3s.f.)

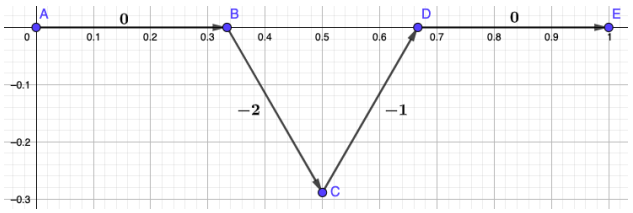


Fig.23: g = {0,-2,-1,0} represented using vectors (Created with GeoGebra)

By using the C_V formula, we can get all the coordinates of the transformed fractal.

Now we can use our code from python to recursively generate {0, -2, -1, 0}

Generation 0 *Generation 1* *Generation 2* *Generation 3*

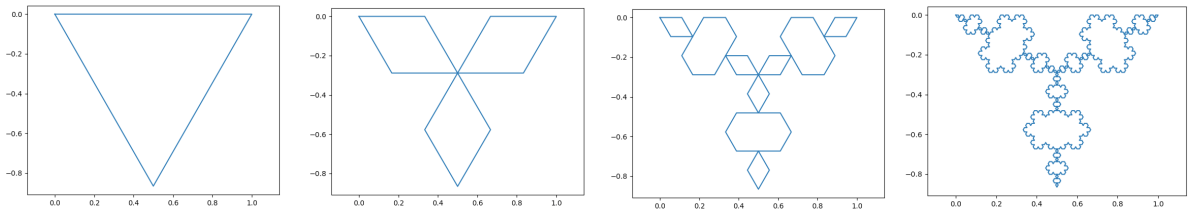


Fig.24: g = g = {0,-2,-1,0} for generation 1 – 4 (Created with Python 3.9.2)

Changing the generator to {0, 2, 0, 0, 1,0} we get: $D = log\ (6)\ /\ log\ (3) \approx 1.63$ (3s.f.)

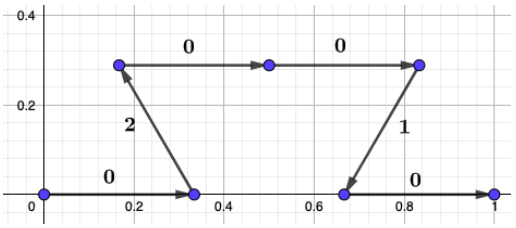
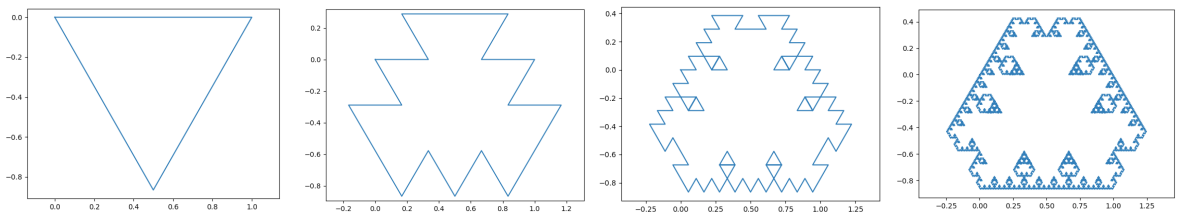


Fig.25: g = {0,2,0,0,1,0} represented using vectors (Created with GeoGebra)



Generation 0 *Generation 1* *Generation 2* *Generation 3*

Fig.26: g = {0,2,0,0,1,0} for generation 1 – 4 (Created with Python 3.9.2)

5.2 Using other initiator functions

Starting with a square (n = 4), we can also use the vectors C_0 to C_{n-1} , as well as their negative reflections (from Fig.27)

With these vectors, we can make an generator function like this {0, 3, 0, 1, 1, 0, 3, 0}

Adding the vectors and scaling it down by ¼, we get:

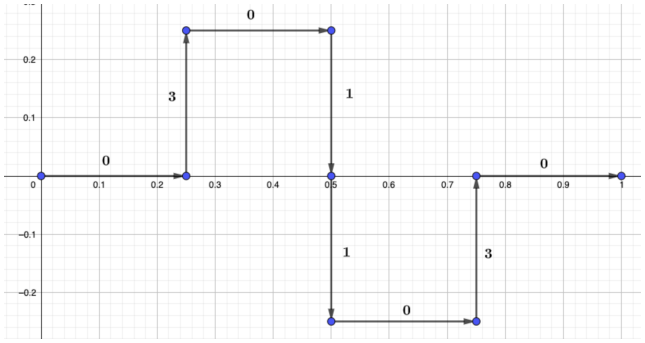


Fig.28: g = {0, 3, 0, 1, 1, 0, 3, 0} represented using vectors (Created with GeoGebra)

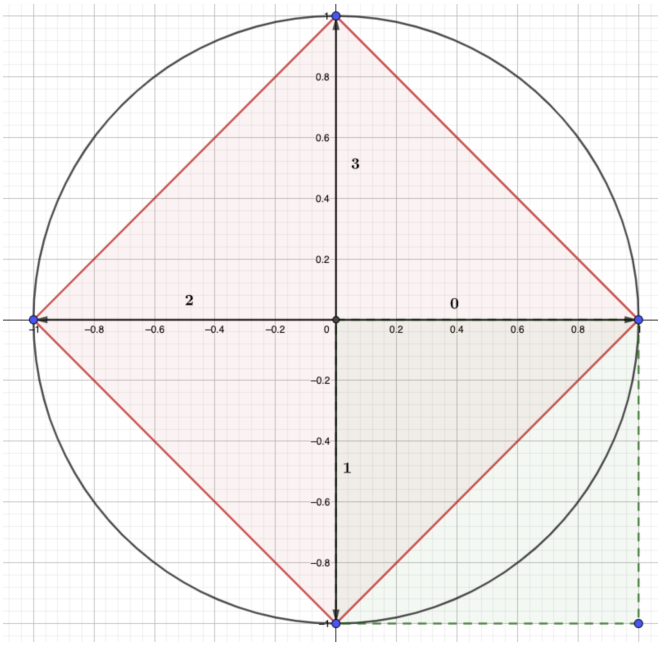
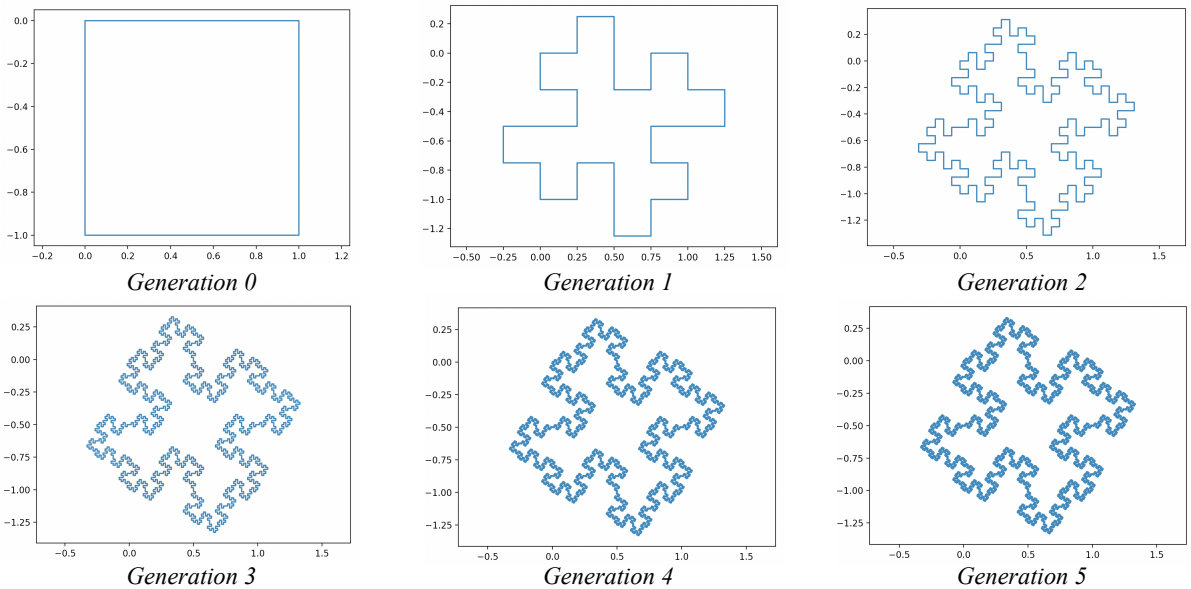


Fig.27: Red:Inscribed regular 4-sided polygon in circle to find vectors 0, 1, 2.3
Green: Translation of vectors to become a closed shape
(Created in GeoGebra)

For the square initiator, using g = {0, 3, 0, 1, 1, 0, 3, 0}, $D = log\ (8) / log\ (4) \approx 1.50$ (3s.f.)



Generation 0 *Generation 1* *Generation 2*

Generation 3 *Generation 4* *Generation 5*

Fig.29: g = {0,3,0,1,1,0,3,0} for generation 1 – 5 (Created with Python 3.9.2)

Starting with a pentagon (n = 5), we can also use the vectors C_0 to C_{n-1} , as well as their negative reflections (from Fig. 30)

With these vectors, we can make an generator function like this {0, -4, -3, -2, -1, 0}

Adding the vectors and scaling by r = ¼

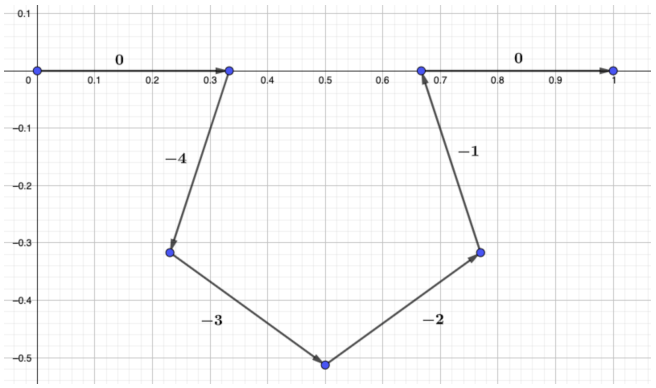
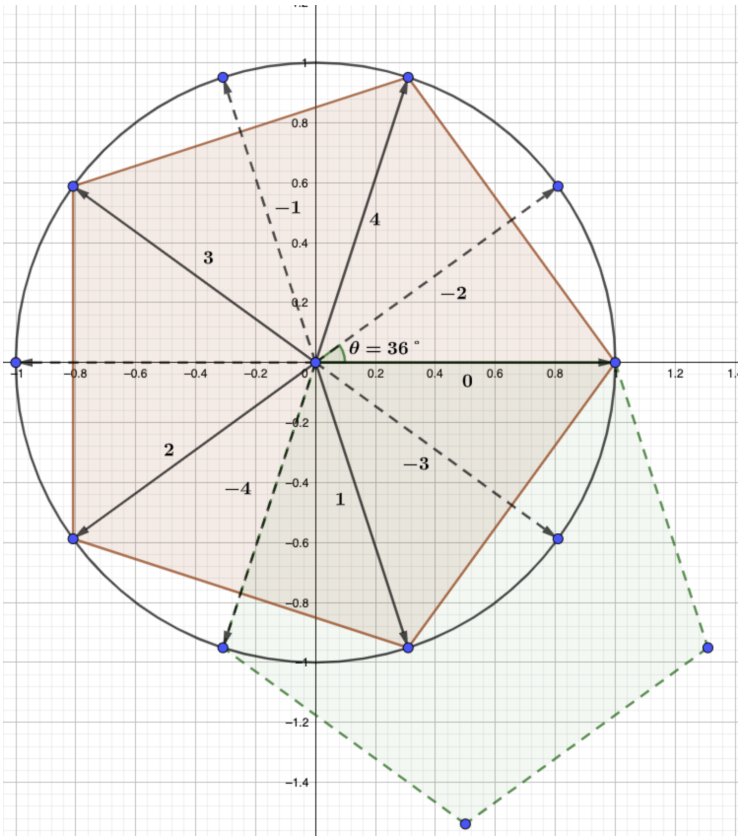
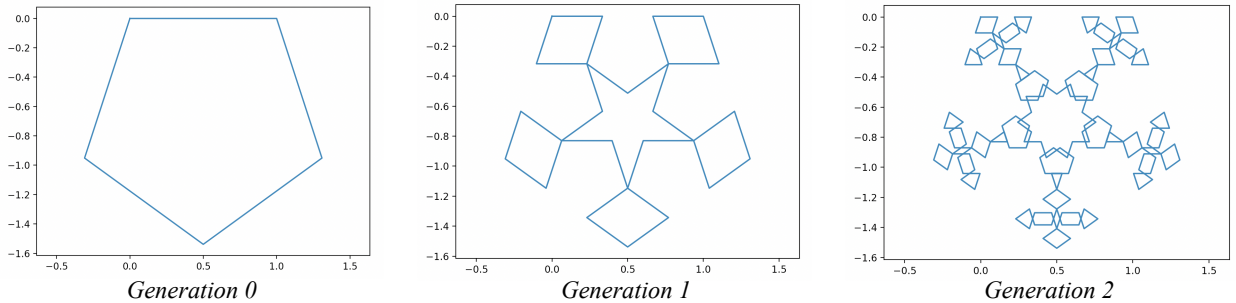


Fig.31: g = {0, -4, -3, -2, -1, 0} represented using vectors (Created with GeoGebra)

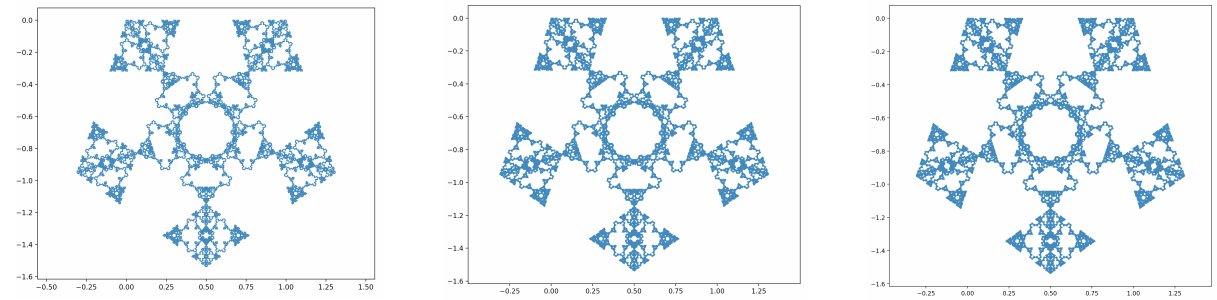
Fig.30: Red:Inscribed regular 5-sided polygon in circle to find vectors 0, 1, 2,3,4
Green: Translation of vectors to become a closed shape
(Created in GeoGebra)



For the pentagon initiator, using g = {0, -4, -3, -2, -1, 0}, $D = log\ (6) / log\ (3) \approx 1.63$ (3s.f.)



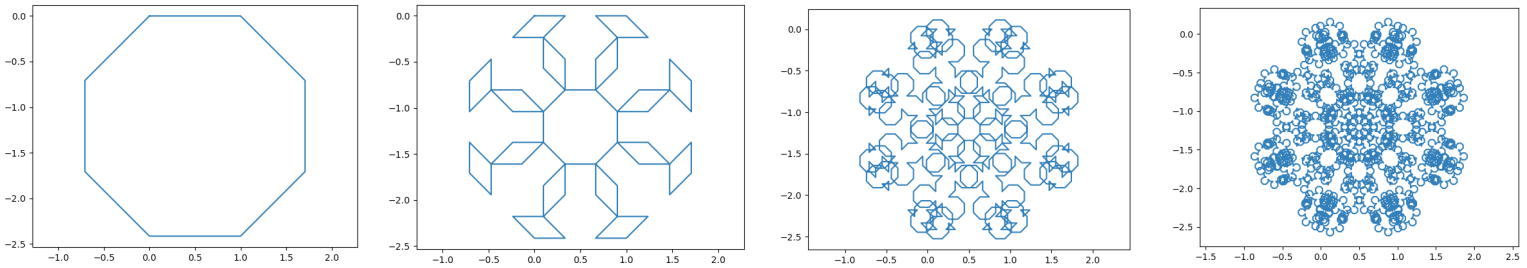
Generation 0 *Generation 1* *Generation 2*



Generation 3 *Generation 4* *Generation 5*

Fig.32: g = {0, -4, -3, -2, -1, 0} for generation 1 – 5 (Created with Python 3.9.2)

For the octagon initiator, using g = {0, -7, -6, -5, -4, -3, -2, -1, 0}, $D = log\ (9) / log\ (3) \approx 2.00$



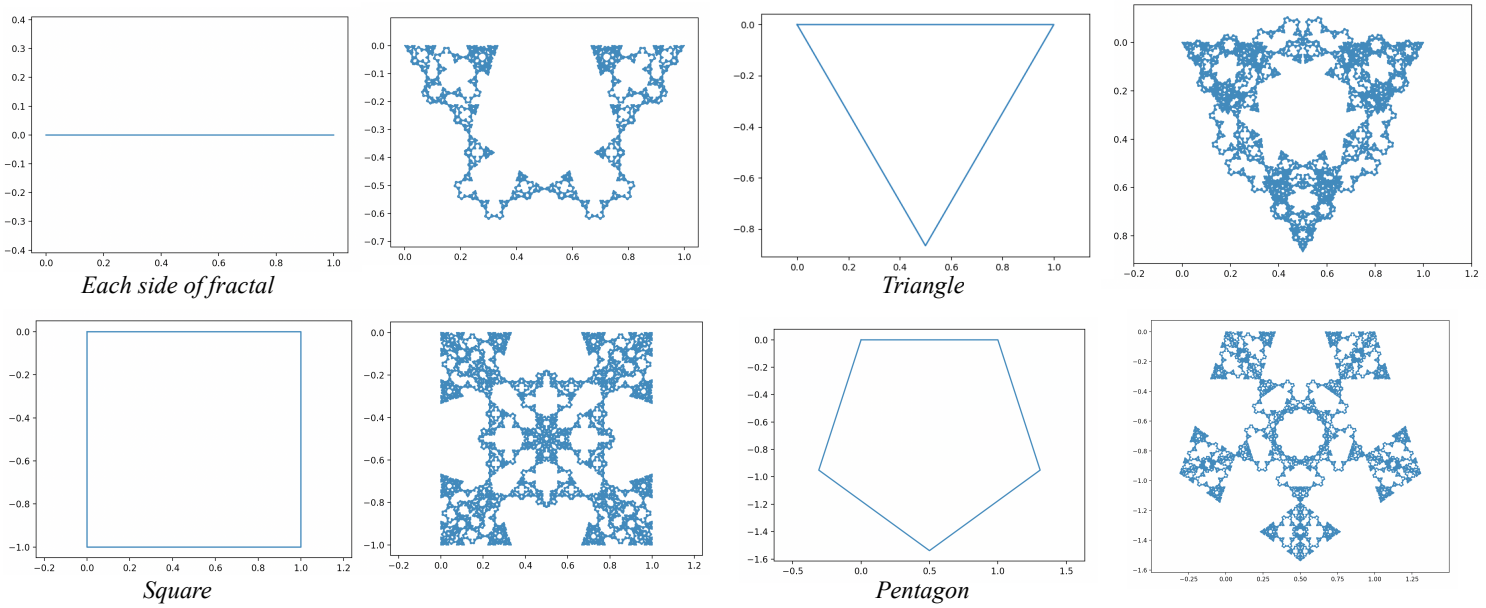
Generation 0 *Generation 1* *Generation 2* *Generation 3*

Fig.33: g = {0, -7, -6, -5, -4, -3, -2, -1, 0} for generation 1 – 3 (Created with Python 3.9.2)

5.3 Applying different initiator to same generator functions

Finally, because we have a recursive relationship between the 2 points of the vectors and how they change due to the generator g, we can create Infinite Koch Fractal (As it does not follow the rules of a n-Koch snowflake) patterns using different *initiators* but the same generator:

Eg. using pentagon *initiator* function, {0, -4, -3, -2, -1, 0} applied to other *initiators*:



Each side of fractal *Triangle*

Square *Pentagon*

Fig.34: g = {0, -4, -3, -2, -1, 0} on different initiators for generation 1 & 5 (Created with Python 3.9.2)

As such, using the math described by this paper, we know how to infinitely generate different types of Koch fractals as well as the general n – Koch Snowflakes for purposes like Batik shirts – combining together the beauty of math and art.

Bibliography

1. Benoît Mandelbrot. (1982). *The fractal geometry of nature*. W.H. Freeman And Company.
2. *Fractals & the Fractal Dimension*. (2019). Vanderbilt.edu. https://www.vanderbilt.edu/AnS/psychology/cogsci/chaos/workshop/Fractals.html
3. *GADM*. (n.d.). Gadm.org. Retrieved March 26, 2021, from https://gadm.org/
4. Shakiban, C., & Bergstedt, J. (2000). Generalized Koch Snowflakes. *BRIDGES 2000 Mathematical Connections in Art, Music, and Science*, *3*(301).
5. Tian, G., Yuan, Q., Hu, T., & Shi, Y. (2019). Auto-Generation System Based on Fractal Geometry for Batik Pattern Design. *Applied Sciences*, *9*(11), 2383. https://doi.org/10.3390/app9112383

Appendix

```
import numpy as np
import matplotlib.pyplot as plt

Rotation_by_60 = np.array([
   [ 1 / 2              ,       -(3 ** 0.5) / 2],
   [(3 ** 0.5) / 2      ,        1 / 2]
   ])

def findB(a, e):
   return a +((e-a) / 3)
def findD(a, e):
   return e -((e-a) / 3)
def findC(a, b):
   return b + np.matmul(Rotation_by_60, (b - a))
def new_gen(points):
   temp_points = np.array([points[0]])

   for i in range(len(points) - 1):
       point_a = points[i]
       point_e = points[i + 1]
       point_b = findB(point_a, point_e)
       point_d = findD(point_a, point_e)
       point_c = findC(point_a, point_b)

       temp_points = np.concatenate([temp_points, [point_b, point_c, point_d,
point_e]] )
   return temp_points

def generate_koch(base, generations):
   points = base

   for _ in range(generations):
       points = new_gen(points)

   return points

base = np.array([[0, 0], [1, 0], [0.5, -(3 ** 0.5) / 2], [0, 0]])  # Triangle

points = generate_koch(base, 4)
print(points)
plt.axis("equal")
plt.plot(*points.transpose())
plt.show()
import math
```

Appendix 1: Python Code for g = { 0, -1, -2, 0}

```
import numpy as np
import matplotlib.pyplot as plt
R = np.array([[0, -1], [1, 0]])
R1 = np.array([[0, 1], [-1, 0]])
def findB(a,i):
   return (3 * a + i) / 4
def findC(b, e):
   return b + np.matmul(R, (e-b))
def findD(b, c):
   return c + np.matmul(R1, (c-b))
def findE(b,h):
   return (b+h)/2
def findF(e,h):
   return e + np.matmul(R1, (h-e))
def findG(e,h):
   return h + np.matmul(R1, (h-e))
def findH(a,i):
   return (a + 3 * i) / 4
def new_gen(points):
   temp_points = np.array([points[0]])
   for i in range(len(points) - 1):
       point_a = points[i]
       point_i = points[i + 1]
       point_b = findB(point_a,point_i)
       point_h = findH(point_a,point_i)
       point_e = findE(point_b,point_h)
       point_c = findC(point_b,point_e)
       point_d = findD(point_b,point_c)
       point_f = findF(point_e,point_h)
       point_g = findG(point_e,point_h)
       temp_points = np.concatenate(
           [temp_points, [point_b, point_c, point_d, point_e, point_f, point_g,
point_h, point_i]])
   return temp_points
def generate_koch(base, generations):
   points = base
   for _ in range(generations):
       points = new_gen(points)
   return points
base = np.array([[0, 0], [1, 0],[1,-1],[0,-1],[0,0]])
points = generate_koch(base, 5)
print(points)
plt.axis("equal")
plt.plot(*points.transpose())
plt.show()
```

Appendix 2: Python code for g = {0, 3, 0, 1, 1, 0, 3, 0}

```
import math
import numpy as np
import matplotlib.pyplot as plt
import operator as operator

R = np.array([[-0.309016994375, -0.951056516295], [0.951056516295,
-0.309016994375]])
R1 = np.array([[-0.309016994375, 0.951056516295], [-0.951056516295,
-0.309016994375]])
R2 = np.array([[0.309016994375, 0.951056516295], [-0.951056516295,
0.309016994375]])
R3 = np.array([[0.809016994375, -0.587785252292], [0.587785252292,
0.809016994375]])
R4 = np.array([[-0.809016994375, 0.587785252292], [-0.587785252292,
-0.809016994375]])

def findB(a,g):
   return (2 * a + g) / 3

def findC(b, f):
   return b + np.matmul(R1, (f-b))

def findD(ima, c):
   return c + np.matmul(R4, (c-ima))

def findE(b,f):

   return f + np.matmul(R, (b-f))

def findF(a,g):
   return (a + 2 * g) / 3

def findIma(b,c):
   return c + np.matmul(R3, (c-b))

def new_gen(points):
   temp_points = np.array([points[0]])

   for i in range(len(points) - 1):
       point_a = points[i]
       point_g = points[i + 1]
       point_b = findB(point_a,point_g)
       point_f = findF(point_a,point_g)
       point_c = findC(point_b,point_f)
       point_e = findE(point_b,point_f)
```

```
        point_imaginary = findIma(point_b,point_c)
        point_d = findD(point_imaginary,point_c)

        temp_points = np.concatenate(
            [temp_points, [point_b, point_c, point_d, point_e, point_f, point_g ]
            ])

    return temp_points


def generate_koch(base, generations):
    points = base

    for _ in range(generations):
        points = new_gen(points)

    return points

base = np.array([
    [0,0],
[ 1.         ,  0.         ],
[ 1+np.cos((-2*np.pi)*(1/8)) ,np.sin((-2*np.pi)*(1/8))],

[ (1 + (np.cos((-2*np.pi)*(1/8)))+ (np.cos((-2*np.pi)*(2/8)))
),(np.sin((-2*np.pi)*(1/8))+ (np.sin((-2*np.pi)*(2/8))))],

[ (1 + (np.cos((-2*np.pi)*(1/8)))+
(np.cos((-2*np.pi)*(2/8)))+(np.cos((-2*np.pi)*(3/8))) ),(np.sin((-2*np.pi)*(1/8))+
(np.sin((-2*np.pi)*(2/8)))+(np.sin((-2*np.pi)*(3/8))))],

[ (1 + (np.cos((-2*np.pi)*(1/8)))+
(np.cos((-2*np.pi)*(2/8)))+(np.cos((-2*np.pi)*(3/8)))+(np.cos((-2*np.pi)*(4/8))) )
,(np.sin((-2*np.pi)*(1/8))+
(np.sin((-2*np.pi)*(2/8)))+(np.sin((-2*np.pi)*(3/8)))+(np.sin((-2*np.pi)*(4/8))))],

[ (1 + (np.cos((-2*np.pi)*(1/8)))+
(np.cos((-2*np.pi)*(2/8)))+(np.cos((-2*np.pi)*(3/8)))+(np.cos((-2*np.pi)*(4/8)))+(n
p.cos((-2*np.pi)*(5/8))) )
,(np.sin((-2*np.pi)*(1/8))+
(np.sin((-2*np.pi)*(2/8)))+(np.sin((-2*np.pi)*(3/8)))+(np.sin((-2*np.pi)*(4/8))))+(
np.sin((-2*np.pi)*(5/8)))],

[ (1 + (np.cos((-2*np.pi)*(1/8)))+
(np.cos((-2*np.pi)*(2/8)))+(np.cos((-2*np.pi)*(3/8)))+(np.cos((-2*np.pi)*(4/8)))+(n
p.cos((-2*np.pi)*(5/8)))+(np.cos((-2*np.pi)*(6/8))) )
```

```
,(np.sin((-2*np.pi)*(1/8))+
(np.sin((-2*np.pi)*(2/8)))+(np.sin((-2*np.pi)*(3/8)))+(np.sin((-2*np.pi)*(4/8))))+(
np.sin((-2*np.pi)*(5/8)))+(np.sin((-2*np.pi)*(6/8))

)],

[ (1 + (np.cos((-2*np.pi)*(1/8)))+
(np.cos((-2*np.pi)*(2/8)))+(np.cos((-2*np.pi)*(3/8)))+(np.cos((-2*np.pi)*(4/8)))+(n
p.cos((-2*np.pi)*(5/8)))+(np.cos((-2*np.pi)*(6/8)))+(np.cos((-2*np.pi)*(7/8))) )

,(np.sin((-2*np.pi)*(1/8))+
(np.sin((-2*np.pi)*(2/8)))+(np.sin((-2*np.pi)*(3/8)))+(np.sin((-2*np.pi)*(4/8))))+(
np.sin((-2*np.pi)*(5/8)))+(np.sin((-2*np.pi)*(6/8)))+(np.sin((-2*np.pi)*(7/8))

)],


[0,0]
])
points = generate_koch(base, 0)
print(points)
plt.axis("equal")
plt.plot(*points.transpose())
plt.show()
```

Appendix 3: Python code for g = {0, -4, -3, -2, -1, 0}

```
import numpy as np
import matplotlib.pyplot as plt

R45 = np.array([[(2**0.5)/2, -(2**0.5)/2], [(2**0.5)/2, (2**0.5)/2]])
R135 = np.array([[-(2**0.5)/2, -(2**0.5)/2], [(2**0.5)/2, -(2**0.5)/2]])
R225 = np.array([[-(2**0.5)/2, (2**0.5)/2], [-(2**0.5)/2, -(2**0.5)/2]])


def findB(a,j):
   return a + ((j-a) / 3)

def findC(a, b):
   return b + np.matmul(R225, (b-a))

def findD(b, c):
   return c + np.matmul(R45, (c-b))

def findE(c, d):
   return d + np.matmul(R45, (d-c))

def findF(d,e):
   return e + np.matmul(R45, (e-d))

def findG(e,f):
   return f + np.matmul(R45, (f-e))

def findH(f,g):
   return g + np.matmul(R45, (g-f))

def findI(a,j):
   return (a + 2 * j) / 3

def new_gen(points):
   temp_points = np.array([points[0]])

   for i in range(len(points) - 1):
       point_a = points[i]
       point_j = points[i + 1]
       point_b = findB(point_a,point_j)
       point_i = findI(point_a,point_j)

       point_c = findC(point_a,point_b)

       point_d = findD(point_b,point_c)
```

```
        point_e = findE(point_c,point_d)
        point_f = findF(point_d,point_e)
        point_g = findG(point_e,point_f)
        point_h = findH(point_f, point_g)

        temp_points = np.concatenate(
            [temp_points, [point_b, point_c, point_d, point_e, point_f, point_g,
point_h, point_i, point_j ]
            ])

    return temp_points


def generate_koch(base, generations):
    points = base

    for _ in range(generations):
        points = new_gen(points)

    return points

base = np.array([
    [0,0],
[ 1.        ,  0.        ],
[ 1+np.cos((-2*np.pi)*(1/8)) ,np.sin((-2*np.pi)*(1/8))],

[ (1 + (np.cos((-2*np.pi)*(1/8)))+ (np.cos((-2*np.pi)*(2/8)))
),(np.sin((-2*np.pi)*(1/8))+ (np.sin((-2*np.pi)*(2/8))))],

[ (1 + (np.cos((-2*np.pi)*(1/8)))+
(np.cos((-2*np.pi)*(2/8)))+(np.cos((-2*np.pi)*(3/8))) ),(np.sin((-2*np.pi)*(1/8))+
(np.sin((-2*np.pi)*(2/8)))+(np.sin((-2*np.pi)*(3/8))))],

[ (1 + (np.cos((-2*np.pi)*(1/8)))+
(np.cos((-2*np.pi)*(2/8)))+(np.cos((-2*np.pi)*(3/8)))+(np.cos((-2*np.pi)*(4/8))) )
,(np.sin((-2*np.pi)*(1/8))+
(np.sin((-2*np.pi)*(2/8)))+(np.sin((-2*np.pi)*(3/8)))+(np.sin((-2*np.pi)*(4/8))))],

[ (1 + (np.cos((-2*np.pi)*(1/8)))+
(np.cos((-2*np.pi)*(2/8)))+(np.cos((-2*np.pi)*(3/8)))+(np.cos((-2*np.pi)*(4/8)))+(n
p.cos((-2*np.pi)*(5/8))) )
,(np.sin((-2*np.pi)*(1/8))+
(np.sin((-2*np.pi)*(2/8)))+(np.sin((-2*np.pi)*(3/8)))+(np.sin((-2*np.pi)*(4/8))))+(
np.sin((-2*np.pi)*(5/8)))],
```

```
[ (1 + (np.cos((-2*np.pi)*(1/8)))+
(np.cos((-2*np.pi)*(2/8)))+(np.cos((-2*np.pi)*(3/8)))+(np.cos((-2*np.pi)*(4/8)))+(n
p.cos((-2*np.pi)*(5/8)))+(np.cos((-2*np.pi)*(6/8))) )

,(np.sin((-2*np.pi)*(1/8))+
(np.sin((-2*np.pi)*(2/8)))+(np.sin((-2*np.pi)*(3/8)))+(np.sin((-2*np.pi)*(4/8))))+(
np.sin((-2*np.pi)*(5/8)))+(np.sin((-2*np.pi)*(6/8))

)],

[ (1 + (np.cos((-2*np.pi)*(1/8)))+
(np.cos((-2*np.pi)*(2/8)))+(np.cos((-2*np.pi)*(3/8)))+(np.cos((-2*np.pi)*(4/8)))+(n
p.cos((-2*np.pi)*(5/8)))+(np.cos((-2*np.pi)*(6/8)))+(np.cos((-2*np.pi)*(7/8))) )

,(np.sin((-2*np.pi)*(1/8))+
(np.sin((-2*np.pi)*(2/8)))+(np.sin((-2*np.pi)*(3/8)))+(np.sin((-2*np.pi)*(4/8))))+(
np.sin((-2*np.pi)*(5/8)))+(np.sin((-2*np.pi)*(6/8)))+(np.sin((-2*np.pi)*(7/8))

)],[0,0]])

points = generate_koch(base, 3)
print(points)
plt.axis("equal")
plt.plot(*points.transpose())
plt.show()
```

Appendix 4: Python code for g = {0, -7, -6, -5, -4, -3, -2, -1, 0}