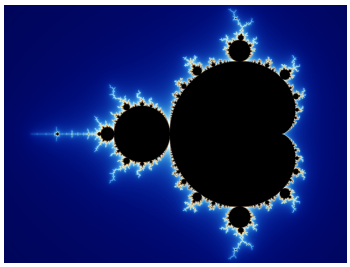
1 Justification

I was first introduced to the topic of fractals a very long time ago within the context of nature. However, I did not develop any rigorous mathematical intution for these bizarre patterns until this term. In my nonlinear dynamics class with Dr. Erdi, we discussed the idea of a fractal's dimension, sometimes referred to as the Hausdorff dimension. We also explored how some fractals, for example the mandelbrot set, are really interesting case studies for chaos - incredibly complex patterns that arise from a defined set of initial conditions. Furthermore, in my Abstract Algebra course with Dr. Intermont, we studied groups, which are intimately tied with the concept of symmetry. I found fractals to be an incredibly beautiful union of symmetry and chaotic behavior, and this exploration is the result of a challenge I imposed onto myself - to generate my own fractals.

(a) Romanesco broccoli [3]

(b) A seashell [2]



(c) The Mandelbrot set [5]

Figure 1: Various examples of fractals

2 Cantor Set

Any mathematician worth his salt is familiar with the Cantor set, often introduced in the beginning of a Real Analysis course. I'll begin this exploration by using the Cantor set as a case study for the Hausdorff dimension, and as the first step towards generating my own fractal by getting accustomed to the recursive nature of their generation.

Figure 2: A representation of how the Cantor set is generated. Plotted in MATLAB by me.

3 Hausdorff Dimension

Let's begin talking about how the dimension of a fractal is defined. The textbook *Understanding Analysis, 2^{nd} Edition* by Stephen Abbott does a wonderful job of illustrating the concept of dimension and defining the dimension of a fractal, so what follows is a brief summary. We can think about the dimension of something (a point, line, square, etc.) by how scaling it affects the output. For example, if we multiply a point by any scalar k, we end up with a point whose size is k^0 times the original. If we scale a line by k, we end up with a line that is k^1 times the length of the original. If we scale a square by k, we end up with k^2 copies of the square. Finally, if we scale a cube by k, we end up with k^3 copies of the cube.

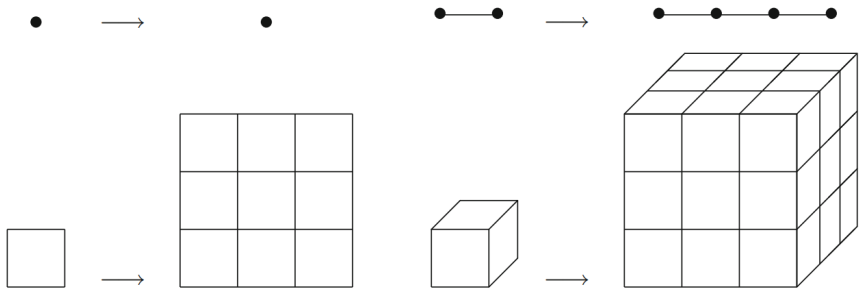


Figure 3: Scaling various geometric objects by a factor of 3. [1]

When we multiply the Cantor set, C, by a factor of 3, we end up with two copies of the original.

	dim	×3	new copies
point	0	→	$1 = 3^0$
segment	1	→	$3 = 3^1$
square	2	→	$9 = 3^2$
cube	3	→	$27 = 3^3$
C	x	→	$2 = 3^x$

Figure 4: The dimension of the Cantor set. Solving for x, we take the logarithm of each side such that $x = \log 2/\log 3 \approx .631$ [1]

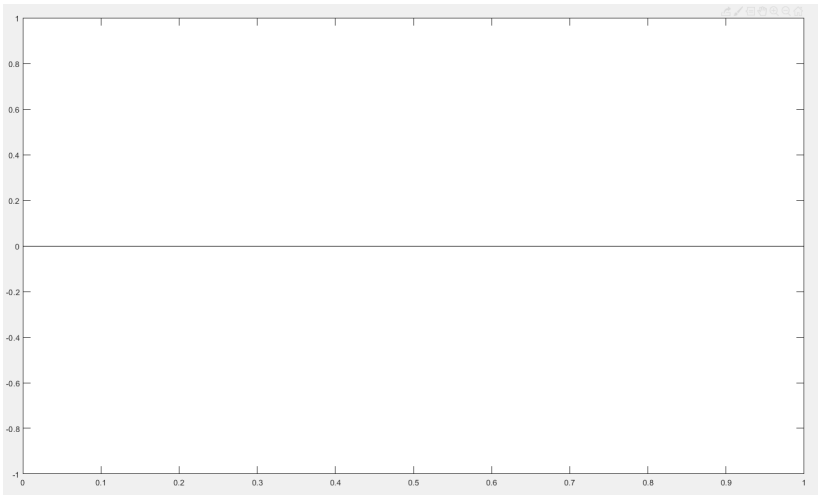
The key result result here is that Hausdorff dimension is not an integer. Instead, it can be written as fraction - which is precisely why the term "fractal" was coined by Benoit Mandelbrot in 1975, "to describe a class of sets whose intricate structures have much in common with the Cantor set." [1]

4 Generating the Cantor set

While it may appear trivially easy to generate the Cantor set, creating it in MATLAB is a necassary first step to getting acquainted with recursive functions. I begin by defining a new function, `cantor(x, y, len)`, that takes an arbitrary coordinate and a length as parameters, then plots a horizontal line.

```
function cantor(x, y, len)
    X = [x; x + len]; %Column vector of X coordinates
    Y = [y; y];       %Column vector of Y coordinates
    plot(X, Y, 'k')   %Draw a line from (x,y) to (x+len, y)
end
```

We exhibit the outcome by calling `cantor(0, 0, 1)`:

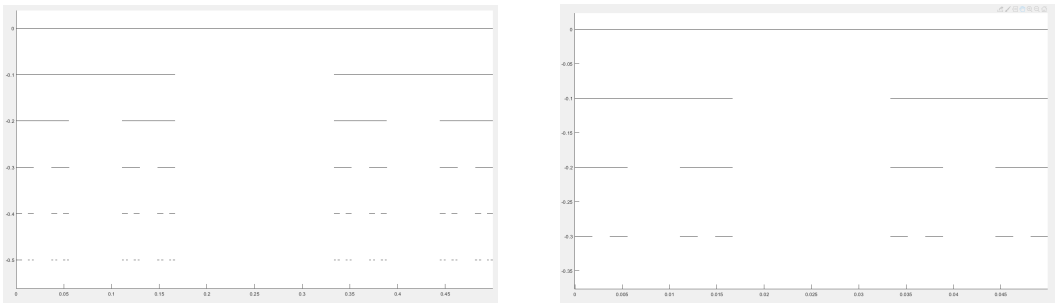


This is a fine and dandy way of drawing a line in MATLAB, however, we know that the Cantor set is generated by removing the middle third at each iteration. So, let's introduced the idea of recursion by building in calls of the `cantor` function within the definition itself - drawing one line that is the first third and a second line that is the final third. Now, because a computer's memory is not infinite, we have to include an exit condition to the recursion so that we don't break the machine that we're running this on. Let's arbitrarily declare that if the length of our line is less than 0.001 then we will stop calling the `cantor` function.

```
function cantor(x, y, len)
    X = [x; x + len]; %Column vector of X coordinates
    Y = [y; y];       %Column vector of Y coordinates
    plot(X, Y, 'k')   %Draw a line from (x,y) to (x+len, y)

    y = y - .1;       %shift the line down, for graphical purposes
    if len > 0.001    %arbitrary exit condition
        cantor(x, y, len/3)           %first third
        cantor(x + 2*len/3, y, len/3) %final third
    end
end
```

I hope that the comments are not too cumbersome, I've decided to include them for the reader who is not familiar with MATLAB syntax or programming in general. It's important to note that setting an arbitrary cutoff makes it so that the number of iterations performed is dependent on the initial length that we use. Consider:



(a) cantor(0, 0, 0.5) (b) cantor(0, 0, 0.01)

Figure 5: Changing the input length changes the number of times that the middle third is removed. We can produce the more aesthetically pleasing image in Figure 2 by removing the axes with the MATLAB command set(gca, 'visble', 'off')

With just a few lines of code, we've been able to (pictorally) represent one of the most important sets in mathematics history!

5 More Fractal Practice

For the sake of readability, I will explain in English/pseudocode how I generated the following fractals, providing the actual MATLAB scripts in an appendix at the end. Let's begin by creating two more fractals using two basic shapes, a circle and a sphere. For the circle, let's have our recursive function draw a base circle and then draw 4 more circles whose centers lie evenly dispersed along the circle's radius, and whose radii are halve those of the previous circle.

Here is what the base circle and one iteration of recursion looks like:

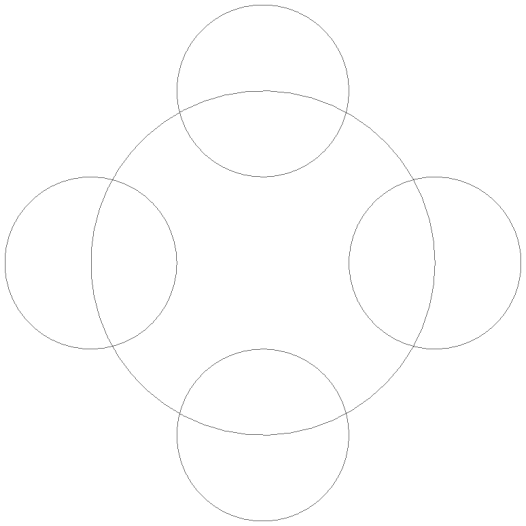


Figure 6: The first step for generating a fractal with circles

Then, adding an arbitrary cutoff to limit the recursion yields the following fractal:

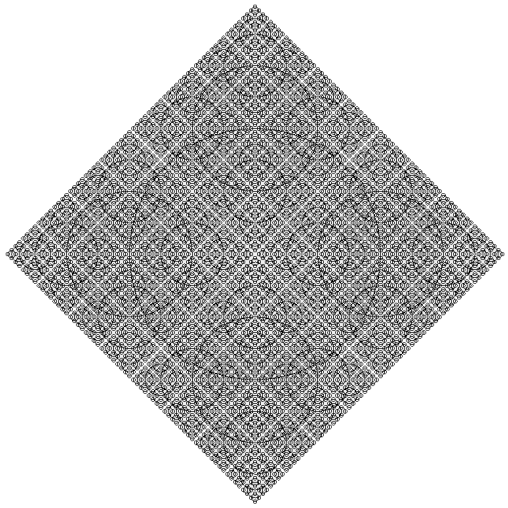


Figure 7: Fractal generated by the function: *recursiveCircles*

We can generate a different pattern by using a similar process, starting with squares, and drawing smaller squares at each of the corners. Observe:

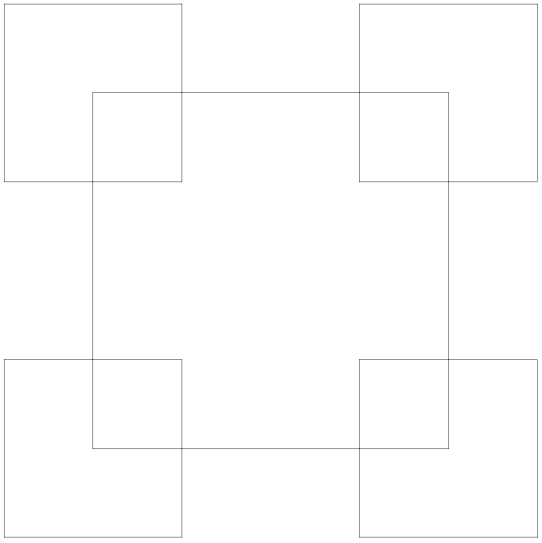


Figure 8: The first step for generating a fractal with squares

And then increasing the number of steps:

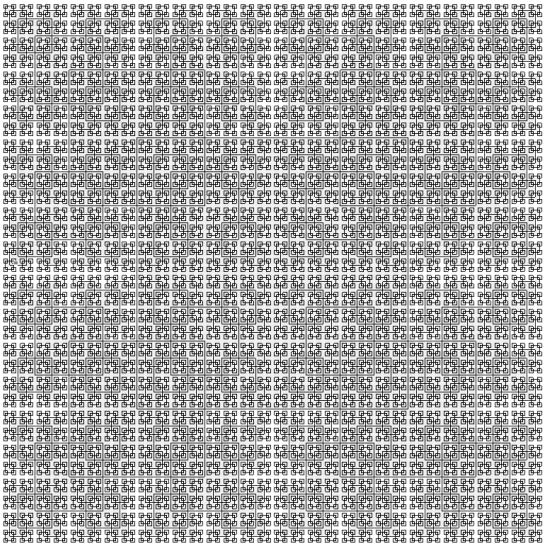


Figure 9: Fractal generated by the function $squareFractals$

Now that we have a basic process for creating reiterative fractals, let's move onto something a little bit more complicated.

6 Koch Snowflake

Another famous fractal is the Koch snowflake.

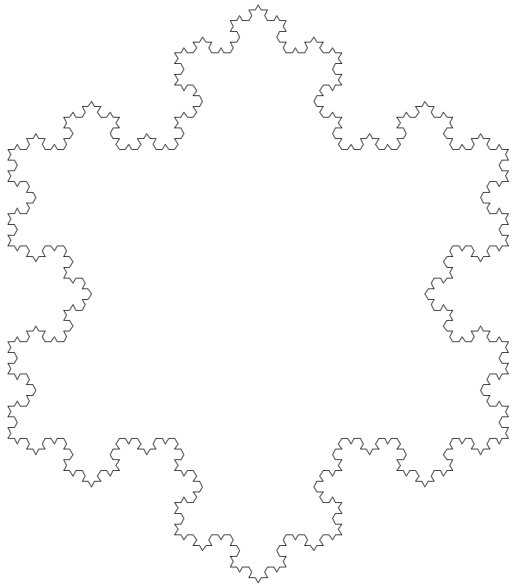


Figure 10: The Koch snowflake [4]

We can generate the entire snowflake by first tackling what happens to a straight line. For every iteration, each straight line adds an equilateral triangle to the middle, whose side lengths are one-third the original line.

Figure 11: One iteration to construct the one third of the Koch snowflake.

This is where I ran into the first difficulty with MATLAB. Whereas for the other patterns generated I could simply recursively call the drawing function, every step of the Koch snowflake requires two 60^o rotations. Because the lines that I have been drawing are not free floating graphical objects - they are encoded as two points in the cartesian plane, a starting point and an endpoint. So, in order to rotate the points about an arbitrary point, I need to first translate the starting point to the origin (translating the end point by the same amount) and then I apply a rotational matrix, found by $\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$, which rotates the points counter clock-wise. Finally, I "undoe" the orginal translation, moving the line back to its starting position, but now in the correct orientation. At

this point, it has become increasingly obvious that MATLAB is not the best language/program to be using when working with generative art, as there is no robust way to rotate a line object where it is. However, there are other programming languages that are designed for visual arts, Processing to give one example.

After making many errors, and having my program behave entirely different than I expected, I was eventually able to generate the one-third of the koch snowflake. Unlike the other functions that I made, this method does not consist of recursive calls, instead I have a `generate` function that creates 4 new lines for every previous current existing line.

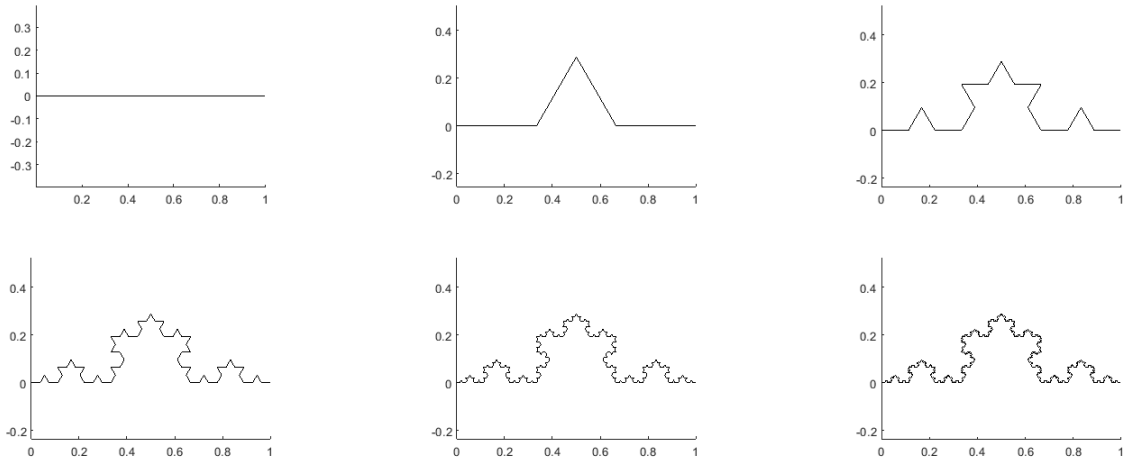


Figure 12: 6 iterations to build the Koch snowflake

On the sixth step there are 1024 distinct lines that are being drawn. However, this is obviously only one-third of the entire Koch snowflake. I was ready to call it quits and be satisfied with this product, because I reasoned that in order to generate the entire structure I would have to rotate and displace all 1024 lines to create the entire snowflake. But, I'm now thinking that I can easily modify the program to start out with 3 lines instead of just one, and I'm pretty sure the generate function that I've already written should do the heavy lifting. Let's try it.

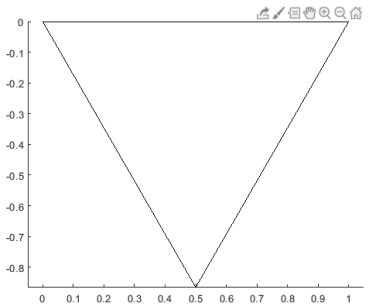


Figure 13: Koch triangle base.

Well, the base triangle was incredibly easy to generate. Given that the line object that I created to draw the Koch snowflake (named `Kochline` in my code) already contained information like start point, end point, and length; I took advantage of the fact that the base shape is equilateral to find the 3^{rd} vertex. All I needed to do was translate the start point x-coordinate by `len`/2 and the y-coordinate by (`len`/2)$*\sqrt{3}$. Now we pray that my `generate` function continues to work:

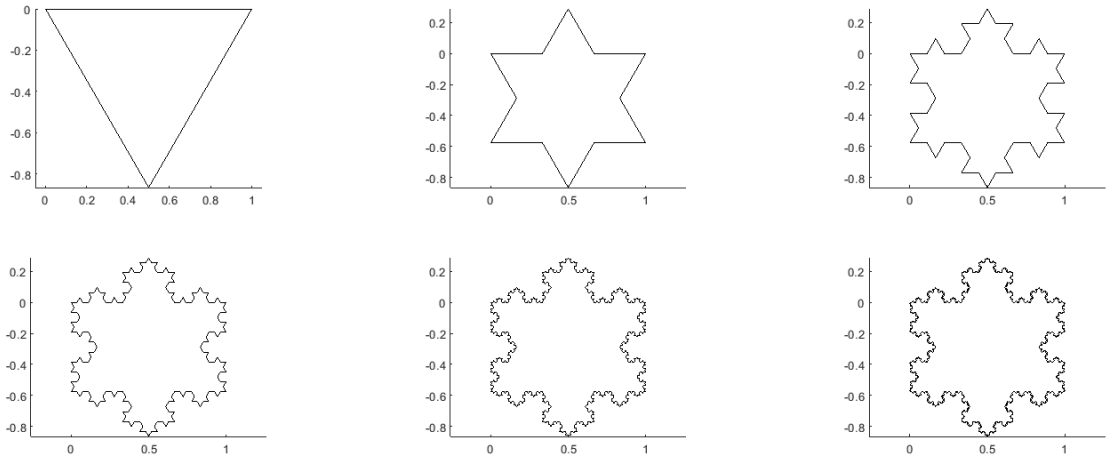


Figure 14: God I love programming.

7 Conclusion

Originally, I wanted to create my own, unique fractal. However, given that this mini-challenge has already become 9-pages long, I'm going to wrap it up here and try and get acquainted with the programming language Processing in order to create more complicated patterns in my own time. I chose to work with MATLAB for this exploration because I have the most experience with it, using it for work on my SIP and to create a simulation project for my nonlinear dynamics course. I really hope you enjoyed this exploration, it was really fun learning more about fractals and how to generate them using code. Attached is a bibliography and an appendix for the code that I wrote in order to complete this little project.

References

[1] Abbott, Stephen *Understanding Analysis, Second Addition, Section 3.1: The Cantor Set, pp. 85-88*

[2] MNN *https://www.mnn.com/earth-matters/wilderness-resources/blogs/14-amazing-fractals-found-in-nature*

[3] WIRED *https://www.wired.com/2010/09/fractal-patterns-in-nature/*

[4] Koch Snowflake *http://gofiguremath.org/fractals/koch-snowflake/*

[5] Mandelbrot set *https://en.wikipedia.org/wiki/Mandelbrot_set*

Appendix

cantor Function

```
function cantor(x, y, len)
    X = [x; x + len]; %Column vector of X coordinates
    Y = [y; y];       %Column vector of Y coordinates
    plot(X, Y, 'k')   %Draw a line from (x,y) to (x+len, y)

    y = y - .1;       %shift the line down, for graphical purposes
    if len > 0.001    %arbitrary exit condition
        cantor(x, y, len/3)           %first third
        cantor(x + 2*len/3, y, len/3) %final third
    end
end
```

recursiveCircles Function

```
function recursiveCircles(x, y, r)
    hold on
    circle(x, y, r) %Draws a circle with center (x, y) and radius r
    if r > 1
        recursiveCircles(x + r, y, r/2)
        recursiveCircles(x - r, y, r/2)
        recursiveCircles(x, y + r, r/2)
        recursiveCircles(x, y - r, r/2)
    end
    hold off
end
```

squareFractals Function

```
function squareFractals(x, y, len)
    square(x, y, len)

    if len > 1
        squareFractals(x + len, y + len, len/2)
        squareFractals(x + len, y - len, len/2)
        squareFractals(x - len, y + len, len/2)
        squareFractals(x - len, y - len, len/2)
    end
end
```

Kochline Class

```
classdef Kochline < handle

    properties
        X
        Y
    end

    properties(Hidden)
        A
        B
        C
        D
        len
        CENTER
    end

    methods
        %KOCHLINE constructs a Kochline object
        %initialPos and finalPos should both be in the form of [x, y]
        function obj = Kochline(initialPos, finalPos)
            obj.X = [initialPos(1); finalPos(1)];
            obj.Y = [initialPos(2); finalPos(2)];
            obj.len = sqrt((obj.X(1) - obj.X(2))^2 + (obj.Y(1) - obj.Y(2))^2);
            obj.A = initialPos;
            obj.D = finalPos;
            obj.CENTER = [sum(obj.Y), sum(obj.Y)]/2;
        end

        %DRAWKOCHLINE plots all the Kochlines in an array
```

```
function drawKochlines(KlineArray)
    %figure
    hold on
    n = length(KlineArray);
    for ii = 1:n
        plotK(KlineArray(ii))
    end
    hold off
    axis equal
    set(gca, 'visible', 'off')
end

%PLOTK plots a single Kochline
function plotK(obj)
    obj.X = [obj.A(1); obj.D(1)];
    obj.Y = [obj.A(2); obj.D(2)];
    plot(obj.X, obj.Y, 'k')
end

%GENERATE Generates four Kochline for every previous Kochline
%in a KlineArray
function next = generate(KlineArray)
    nLength = length(KlineArray)*4;
    n = length(KlineArray);
    next = Kochline.empty(0, nLength);
    for ii = 1:n
        jj = ii*4;
        thisObj = generatePoints(KlineArray(ii));
        a = thisObj.A;
        b = thisObj.B;
        c = thisObj.C;
        d = thisObj.D;

        next(jj - 3) = Kochline(a, b);
        next(jj - 2) = Kochline(b, c);
        next(jj - 1) = Kochline(b, c);
        next(jj) = Kochline(c, d);

        rotccw(next(jj - 2));
        rotcw(next(jj - 1));

        next(jj-2).A = next(jj-2).A + (next(jj - 1).D - next(jj - 1).A);
        next(jj-2).D = next(jj-2).D + (next(jj - 1).D - next(jj - 1).A);
    end
end
```

```
        %GENERATEPOINTS generates the points needed for the next generation
        %of Kochlines
        %Only call after the object has been rotated
        function obj = generatePoints(obj)
            translation = (obj.D - obj.A)/3;
            obj.B = obj.A + translation;
            obj.C = obj.B + translation;
        end

        %ROTCW rotates a Kochline 60degrees clockwise
        function rotcw(obj)
            endPoint = (obj.A - obj.D)'; %Translate to the origin
            rotMatrix = [cos(-pi/3), -sin(-pi/3); sin(-pi/3), cos(-pi/3)];
            newEndPoint = rotMatrix*endPoint;
            obj.D = newEndPoint' + obj.D; %Translate back to the start point
        end

        %ROTCCW rotates a Kochline 60degrees counterclockwise
        function rotccw(obj)
            endPoint = (obj.A - obj.D)'; %Translate to the origin
            rotMatrix = [cos(pi/3), -sin(pi/3); sin(pi/3), cos(pi/3)];
            newEndPoint = rotMatrix*endPoint;
            obj.D = newEndPoint' + obj.D; %Translate back to the start point
        end

        %CREATETRIANGLE initialises the Koch base from one Kochline
        function KlineArray = createTriangle(obj)
            KlineArray = Kochline.empty(0,3);
            KlineArray(1) = obj;
            xtrans = (obj.len)/2;
            ytrans = sqrt(3)*(obj.len)/2;
            thirdVertex = [obj.A(1) + xtrans, obj.A(2) - ytrans];

            KlineArray(2) = Kochline(obj.D, thirdVertex);
            KlineArray(3) = Kochline(thirdVertex, obj.A);
        end


    end
end
```