University of California, Davis

# Fractal Package

STA242_Final Project

git@bitbucket.org:gliang1/sta242_project.git

Guannan Liang/912505172
Luhong Pan/912428000
Di Zhang/912458216

# FRACTAL PACKAGE

Guannan Liang[1], Luhong Pan[2] and Di Zhang[3]

*Department of Statistics, University of California, Davis, CA, 95616, USA*

**Abstract**

This report intended to introduce a package that containing the functions in R programming language that realize and visualize the basic and extended sets of fractal in the mathematical area, which could have educational value and artistic value.

## 1. Fractal Introduction

A fractal is a natural phenomenon or a mathematical set that exhibits a repeating pattern that displays at every scales. The fractal sets all have the characteristic of self-similarity, which could be exact-self-similarity, quasi self-similarity, statistical self-similarity and qualitative self-similarity. No matter in what case, a fractal set has fine or detailed structure at arbitrarily small scales.

Fractal theory changes our views in at least three aspects: (1) many seeming irregular forms in nature have hidden regularity and could be built on fractal set; (2) many seeming random forms in the past could be explained by fractal in regular way; (3) the theory of fractal dimension provide a new measurement of complicated real life issues.

There are different techniques to generate fractal sets, including recursion, iterated function system, L-systems, escape-time-fractals, finite subdivision and random fractals. In our project, we collected and optimize the algorithm of existing fractal sets, visualized them, and extended new algorithm (transformation) based on that. Finally we created a package containing the algorithm and visualization for educational or artistic purpose.

## 2. Package Introduction

This package included sets and their extension set. The main goal of the package is applying different algorithm to generate sets of different dimensions and visualize them. There are four main families of sets, one external method and two revised sample sets to display here.

### 2.1 Julia Set

#### 2.1.1 Introduction

Julia set and Mandelbrot set are both generated from iteration function

$$Z_{i+1} = Z_i^2 + c$$

[1] Guannan Liang: gnliang@ucdavis.edu

[2] Luhong Pan: plhpan@ucdavis.edu

[3] Di Zhang: ddizhang@ucdavis.edu

A common way to display the sets is to plot them, pixar by pixar, on the complex plane. For Mandelbrot set, each point corresponds to a c in f(Z), and $Z_0$ is set to be zero. For Julia set, each point corresponds to an initial value $Z_0$, given c. The color of each point corresponds to the iteration before f(z) exceeds a certain value. The technique used here is "Escape Time Algorithm". For every given point in Mandelbrot set, we can generate a Julia set with c equals to that point.

### 2.1.2 Highlight

There is already a Julia Set package in the website CRAN (Julia) but when we looked into the functions of that package, we found more effort could be done to make improvement.

First, there is one mistake in the Julia Package causing the color display of Julia and Mandelbrot sets to be inaccurate. Specifically, the point sets where the objective function converges and diverges are wrongly drawn in the same color. Second, the speed of iteration is barely satisfactory. The implementation of Julia and Mandelbrot set is very computational intensive. Its algorithm, the escape time algorithm, relies greatly on loops. Since running loops isn't R's strength, it would be desirable to implement this part in low-level languages.

In the functions provided in our package, the color display problem is fixed. Also, the implementation of escape time algorithm is moved to C language, which has increased the speed by about 100 times.

Additionally, a user-interactive function is added which allows user to get the Julia set derived from a specific point in Mandelbrot set. An illustration examples is provided in the figure below in Figure 1.

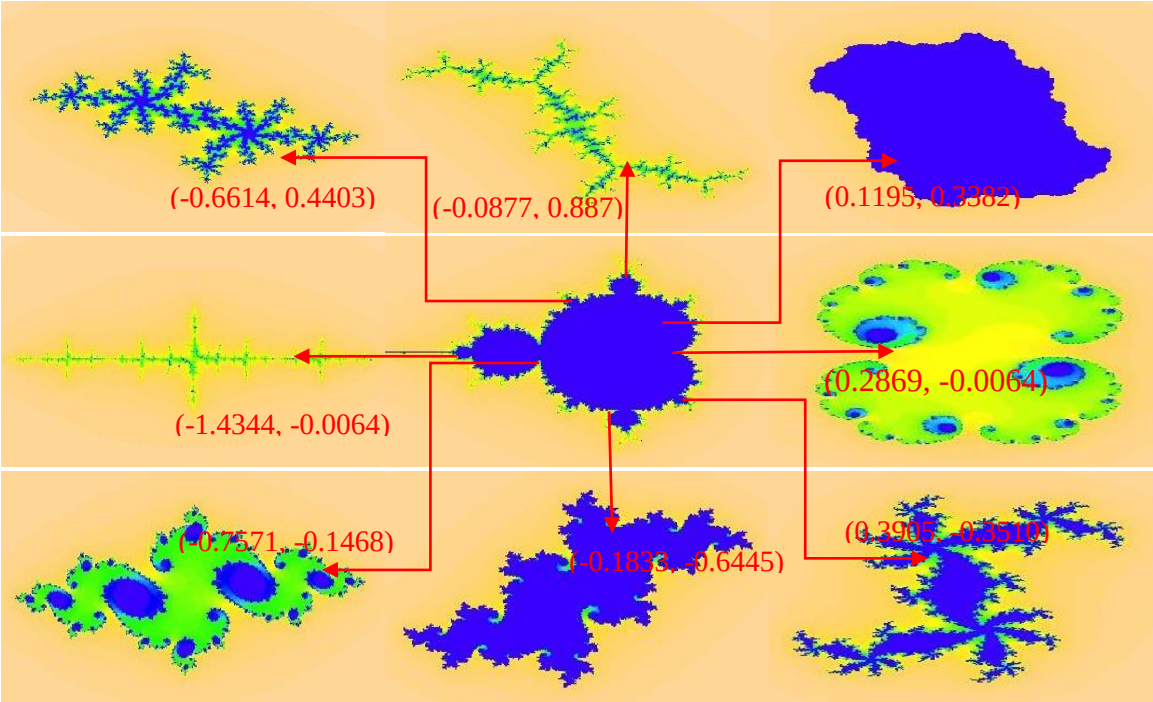


FIGURE 1. Julia Set Examples

In the example in Figure 1. We choose seven different points from the Mandelbrot set in the middle picture and then generated seven Julia Set using the x+yi observed in the graph as initial condition.

2.1.3 Functions

- julia(constant, ...): generate a julia set given c.
- mandelbrot(...): generate a mandelbrot set.
- plot.Julia(): plot a julia object.
- plot.Mandelbrot(): plot a mandelbrot object.
- getJulia(mandelbrot): interactively generate a Julia set from a point in Mandelbrot set.

2.2 Koch Curve Family

2.2.1 Koch Curve

Koch Curve is a normal seeing set but in the fractal theory but strangely we only found an example of generating a given dimension sets with points. Therefore, we designed the algorithm and wrote the function all by ourselves to allow users to decide the number of dimension. The technique we used here is recursion. The first 5 dimension sets were shown in Figure 2.

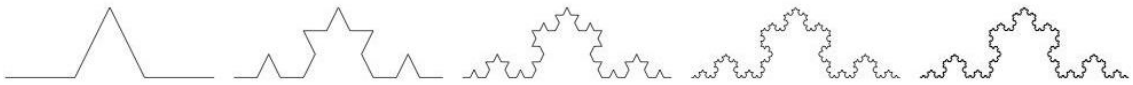

FIGURE 2. Koch Curve Sets Transformation

Koch Curve is based on a straight line and the first dimension is showed in the first picture of Figure 2. which has describe the pattern: separate the line into three segments and replace the middle one with a broken line of an incomplete triangle. For the following dimension of sets we just repeat the pattern on each segment. We denote the first point of the pattern as (0, 0). The way we document the pattern is the distance of each point to (0, 0) and the angle between the line to the positive x-coordinate, according to which we could create function to generate the next dimension sets.

2.2.2 Koch Snowflake

Koch Snowflake is the transformation based Koch Curve. Instead of generating fractals on a straight line, it starts from a triangle. All the technique and algorithm used in Koch Curve is applied here in generating Snowflake.

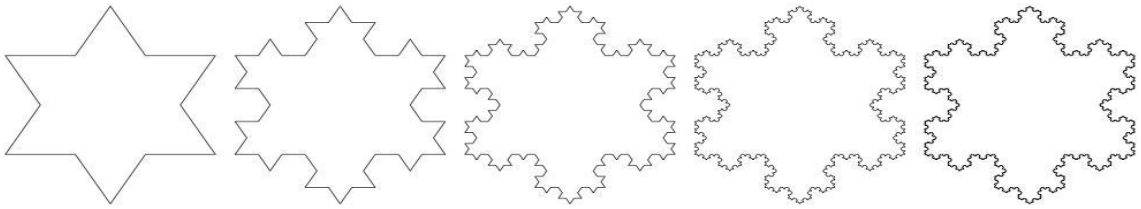

FIGURE 3. Koch Snowflake Transformation

2.2.3 User-created Sets

Another creative aspect of our package about the Koch Curve Family is to allow user to define the initial line or figure and apply fractal theorem to create self-similarity images. When you call pattern() function in the package, you can use the mouse to point some points you want to create a initial images for us to iterate. Several examples could be found in Figure 4. with the initial pattern on the left and finished sets displayed on the right. In the first example we use 9 points to create a shape of heart, our function generate clusters of hearts based on the fractal theorem. The second example is to use 4 point to generate a broken line and a cloud like figure generated. And the last one, four points created broken line generated a Hog's Head.

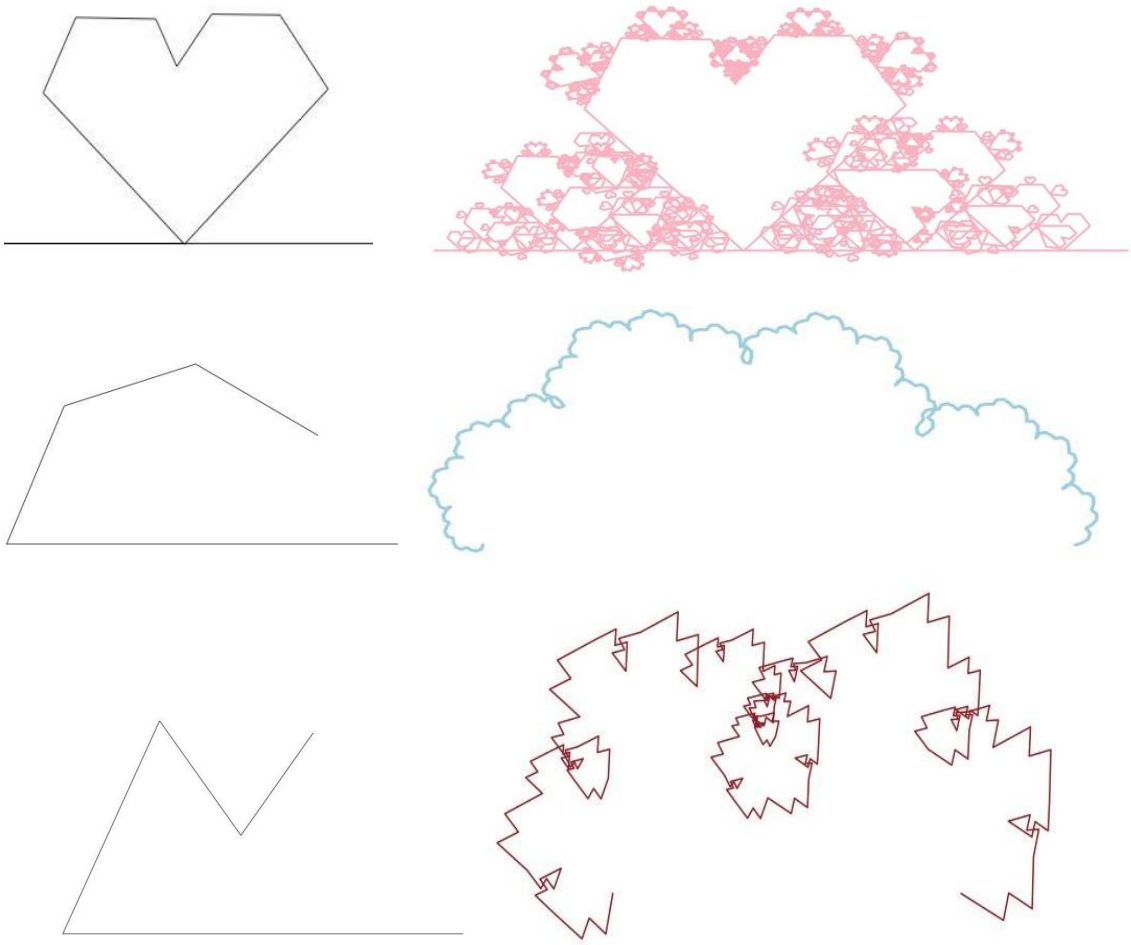

FIGURE 4. Samples of User-created Sets: Heart, Cloud and Hog's Head

2.2.4 Function

- koch(x1, x2, y1, y2, itr): generate a classic Koch Curve with specific iteration time on a given segment.

- kochSnowFlake(a, itr): generate a classic Koch Snowflake with length a on each edge with specific iteration time.
- kochPlus(x1, x2, y1, y2, itr, ptn): transform a segment with a given pattern in the Koch way. In other words, generate a "General" Koch curve.
- pattern(): define the pattern used to generate General Koch Curve.
- plot.kochPlusCurve(): plot the set
- plot.kochSnowFlake(): plot the set

## 2.3 Sierpinski Triangle Family

### 2.3.1 Sierpinski Triangle

There is also a package of Sierpinski Triangle in CRAN that works very well in creating Sierpinski Triangles allowing user to define the angles to a triangle and then generate sierpinski triangle sets according to different rule: by the center (upper one) or the focus (bottom one) as shown in Figure 5.

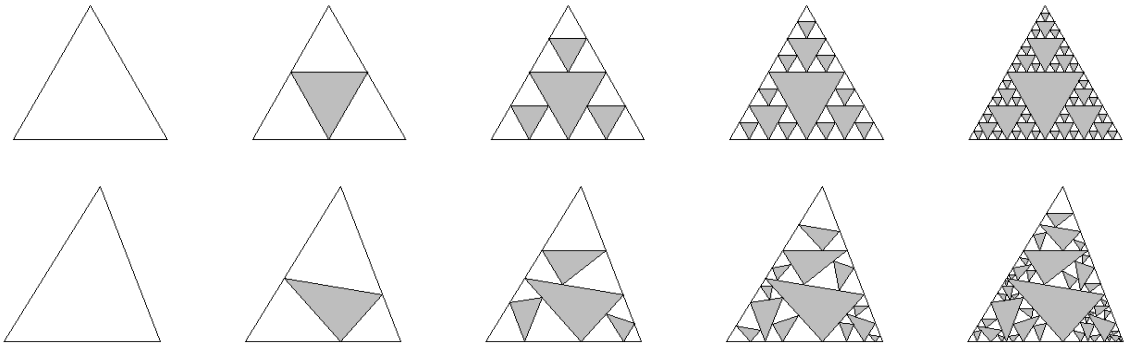

FIGURE 5. Samples of Sierpinski Triangle

### 2.3.2 Arrowhead Curve

The fractal theory could be used to generate new sets under the self-similarity. What we did here is to change realize a new sets in this family:"Arrow Head". The set was derived from the sets of "Sierpinski Triangle" and "Koch Curve" introduced above. For "Sierpinski Triangle", the set keep generating triangles to the outside layer and for "Koch Curve" it is only a line keeping bending at the middle of the lines. "Arrow Head" keeps the triangle sharp of the sets while using only a line to do the self-similar transformation. The transformation could be easily explained by Figure 6.

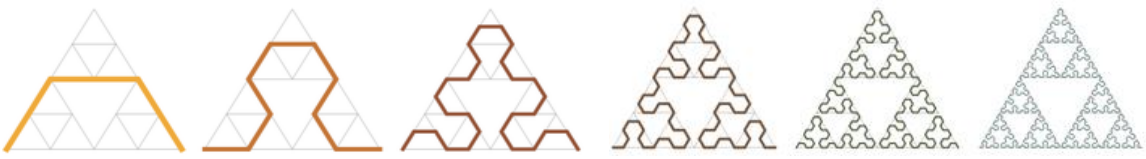

FIGURE 6. Sierpinski Triangle to Arrowhead Transformation

For now, we did not find the existing command to realize it so we designed algorithm and wrote the code all by ourselves. The technique here we used is iteration to generate every dimension of the sets. The first 5 dimension sets were shown in Figure 6. We grasped the initial coordinates of the four points in the first picture. The second dimension "Arrow Head" sets were formed by three parts, and we rotated the four points to 2*pi/3 degree and moved it to get the left bottom part of the second picture, translate all the points to get the middle upper part of the second and rotated -2*pi/3 degree to get the right bottom part. For the third dimension set we just repeat the same process on the second dimension set and for the nth dimension set we use the same method based on the (n-1)th dimension set.

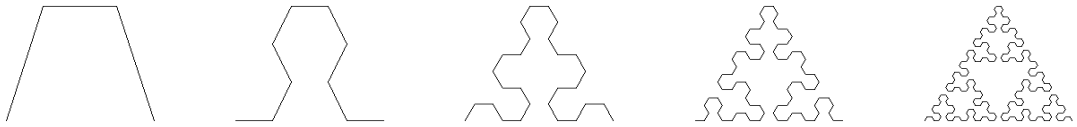

FIGURE 6. Arrow Head Sets Transformation

2.3.3 Functions

- ArrowheadCurve(): user can define the number of iterations to general an Arrowhead set of certain dimension. The return value is two vectors of numbers indicating the coordinates of the ordered points.
- plot(): allow user to visualize the generated sets.

2.4 Hilbert Curve Family

2.4.1 Introduction

The fourth fractal sets we explored is the family of Hilbert Curve, which was named after German mathematician David Hilbert who discovered it in the early 1900's. One of the interesting aspects about the sets in Hilbert family is that it is a "space-filling" curve: it literally covers every point in a square.

The generation of Hilbert curve starts from the pattern in the first picture of Figure 7. First we divided the squares into four small ones equally. The starting pattern is a "U" shape broken line connect the center of the four small squares. Each of the 4 squares has been divided into 4 more squares. To generate the Hilbert Curve of second dimension, the U shape broken line shrunk to half its original size and copied itself by four to fill the four squares. In the top left, it is simply copied and moved. In the top right, it is flipped horizontally. In the bottom left, it is rotated 90 degrees clockwise, and in the bottom right, it is rotated 90 degrees counter-clockwise. The 4 pieces are connected with 3 segments displayed in red in Figure 7. The curve of next dimension we based on the previous one by repeating the same process.

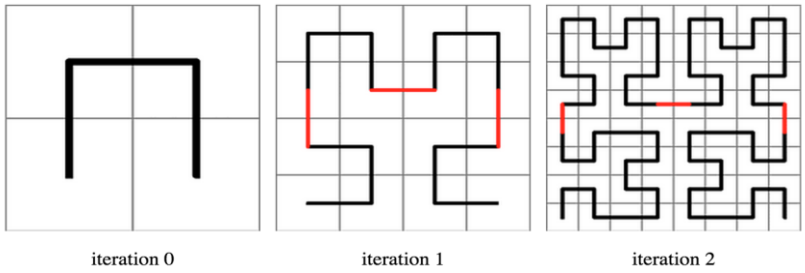


FIGURE 7. Iteration of Hilbert Curve

2.4.2 Function Improvement

In our package we enhanced the function by allowing the shape of the block to be rectangles. The first, third and fifth dimensions of Hilbert Curve sets were shown in Figure 8. Below.

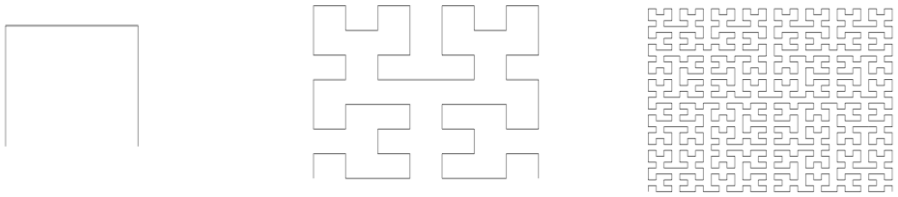

FIGURE 8. Samples of Hilbert Curve

Based on the enhanced function, we notice that as long as the beginning and ending point of U broken line is settle, the initial pattern could be more flexible. We created new transformed Hilbert Curve by setting the beginning and ending points while randomly generating the second and third point to generate the sets, the graphs were shown in Figure 9 as the initial line is irregular.

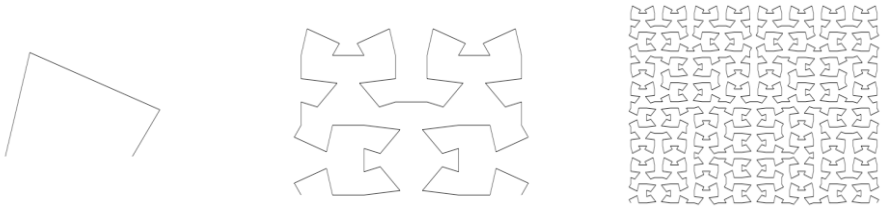

FIGURE 9. Samples of Enhanced Hilbert Curve

Finally, we applied the user-defined pattern model into the Hilbert family curve and allowed user to use mouse to define the four points interactively while generating the sets according to the same fractal theory. The examples were shown in Figure 10.

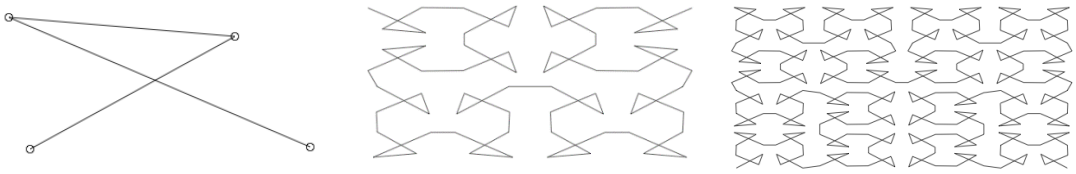

FIGURE 10. Samples of User-Created Hilbert Curve

2.4.3 Functions

- CreateHCF(): The function create the origin Hilbert Curve matrix.
- hilbert.curve():The function is to growth the Hilbert curve n steps.
- plot.Hilbert(): This function plots the Hilbert matrix.

2.5 New Method of Displaying Incursive Sets

In this section, we introduced a new method to display the fractal pattern. We used the example in Figure 10. to explained it. In the three black and write pictures we connected the points generated by fractal iteration function; while in the last warm color picture, we apply the method of assigning the color to each point by the distance between the point and the previous one. Therefore, the same color indicated that the similarity in distance.

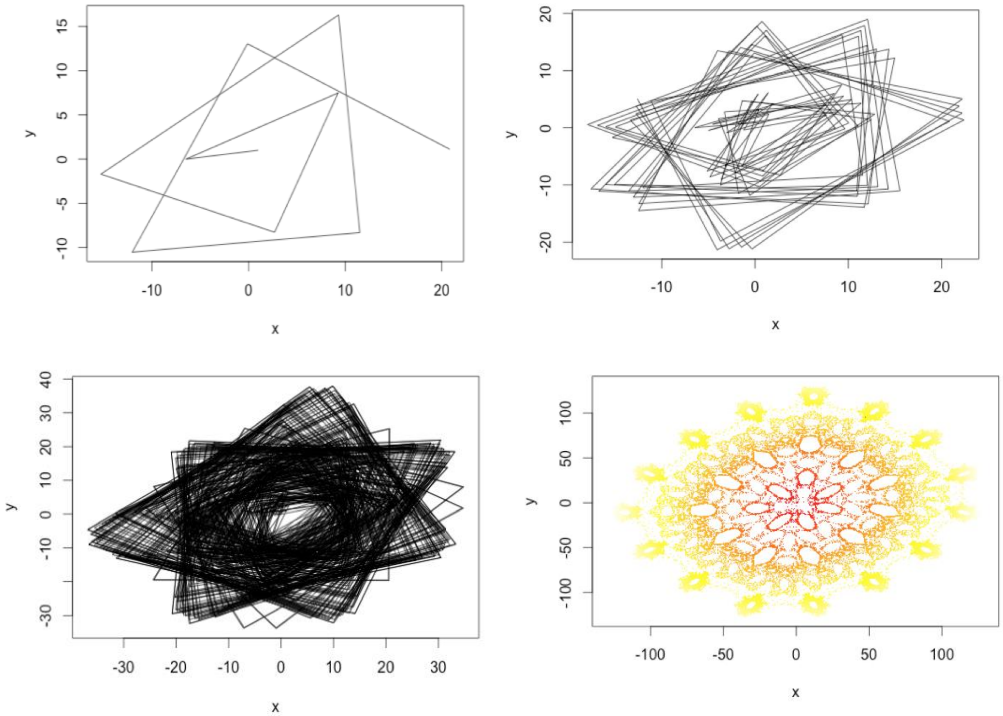


FIGURE 10. Samples of User-Created Hilbert Curve

2.6 Other Sets Revised

Finally, we also revised two interesting samples of plants related sets. One is Barnsley Fern (shown on the left of Figure 11.), which self-generates a fern like graphs by repeating itself as the leaves and then branches using L-system. The other is Flower with the similar generation theory but use the chaos game methods.

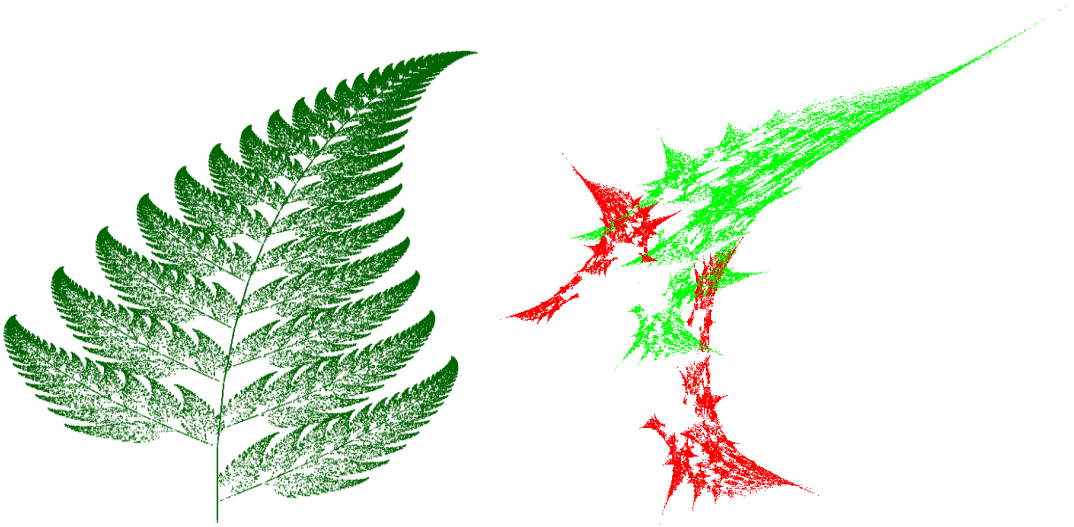

FIGURE 11. Barnsley Fern and Flower

Functions involved in these two examples are:
BarnsleyFern(): will display the fern example.
Flower(): will display the flower example.

## 3. Conclusion and Further Discussion

3.1. Conclusion

In this project we created a package to realize and visualize some fractal sets. The creative aspects of this work includes three parts.

3.1.1 Extending and Exploring Theory

First is to flexibly extend and explore the fractal theory: the creation of function of Arrowhead is an example to show the theory extension as it is a combination of two kinds of fractal sets. The transformation of the initial pattern in the Hilbert and Koch Curve also shows the application of theory flexibly.

3.1.2 Creating or Improving Codes

The second is to initially create R functions for sets that do not have existing codes such as Koch Snowflake and Arrowhead Curve, or the existing codes need great improvement such as Julia Set, Hilbert Set and Koch Curve. By rewriting the function, we allow user to choose the dimension of the sets that they are interested in or would like to display and they can see how the sets generate by themselves.

### 3.1.3 Enhancing Function

The third aspects is function enhancement: we created an interactive function allowing users to define the initial pattern for generation. This allows user to apply the fractal theory to create infinite new sets in the different families. Just as we have shown in the Koch Curve family and the Hilbert family.

After the research jobs, we included as much fractal sets as we could to create the package. Since it could create delicate graphs and self-define graphs, it is flexible and of artistic and illustrational value.

## 3.2. Further Discussion

The fractal theory is very difficult and profound, leaving lots of explore. For the time limit, we cannot finish all.

### 3.2.1 Algorithm

As we have discussed before there are lots of algorithm could be used to generate fractal sets. To be more precise one sets could be generated by different techniques. If we could apply all the techniques that could be used to generate a certain set, the algorithm could be improved to the largest extend.

The most common techniques we used here in our package are recursion, iterated function, L-systems (for Barnsley Fern) and escape-time-fractals (Julia Sets). For finite subdivision and random fractals we did not apply and were not very familiar with. We could explore more about that method if we have more tome.

### 3.2.2 Embellishment

As an enhancement of artistic value, there are two aspects that we could improve our package. We could increase the range of colors to make the graph of sets look better and use more method to plot to make the function run faster: since R does not provide enough color range and faster function to graph complicated figure, we are limited in providing more sophisticated fractal graph. However, we could explore external methods and software to work with R together to visualize the sets in a better way.

### 3.2.3 Higher Level Fractal Sets

3D fractal sets and plots is another direction to extend the artistic value of fractal sets and the algorithm could be more complex and challenging for further exploration.

## Appendix: Selected Code

```
##  Julia Set
library(devtools)
setwd( "/Users/Neil/Dropbox/242FinalProject/Fractal/成品")
dyn.load("JuliaMandelbrot.so")

julia = function(constant, len = 4, pix = 400, center = 0, maxItr = 50, radius = 4)
{
 result = .C("julia", as.integer(pix), as.double(Re(center)), as.double(Im(center)),
          as.double(len), as.double(Re(constant)), as.double(Im(constant)),
          as.integer(numeric(pix^2)), as.integer(maxItr), as.double(radius))[[7]]
 #return the matrix
 julia = list( matrix = matrix(result, pix, pix), center = center, constant = constant,
           maxItr = maxItr, escapeRadius = radius, plotLength = len)
 class(julia) = c("Julia", class(julia))
 julia
}

mandelbrot = function(pix = 400, len = 4, center = 0, maxItr = 50, radius = 4)
{
 result = .C("mandelbrot", as.integer(pix),
          as.double(Re(center)), as.double(Im(center)), as.double(len),
          as.integer(numeric(pix^2)), as.integer(maxItr), as.double(radius))[[5]]

 mandelbrot = list( matrix = matrix(result, pix, pix), center = center,
          maxItr = maxItr, escapeRadius = radius, plotLength = len)
 class(mandelbrot) = c("Mandelbrot", class(mandelbrot))
 mandelbrot
}

plot.Julia = function(x, ...)
{
 mat = x$matrix
 len = x$plotLength

 image(t(mat[nrow(mat):1, ]), col = rev(topo.colors(x$maxItr)), axes = FALSE)
 axis(1, at = seq(0, 1, by = 0.2),
    labels = as.character(seq(Re(x$center) - len/2, Re(x$center) + len/2, length.out = 6)))
 axis(2, at = seq(0, 1, by = 0.2),
    labels = as.character(seq(Re(x$center) - len/2, Re(x$center) + len/2, length.out = 6)))
 title(paste0("Julia set with constant", Re(x$constant), "+", Im(x$constant), "i"))
}

plot.Mandelbrot = function(x, ...)
{
 mat = x$matrix
 len = x$plotLength
```

```
  image(t(mat[nrow(mat):1, ]), col = rev(topo.colors(x$maxItr)), axes = FALSE)
  axis(1, at = seq(0, 1, by = 0.2),
      labels = as.character(seq(Re(x$center) - len/2, Re(x$center) + len/2, length.out = 6)))
  axis(2, at = seq(0, 1, by = 0.2),
      labels = as.character(seq(Re(x$center) - len/2, Re(x$center) + len/2, length.out = 6)))
  title("Mandelbrot Set")
}


getJulia = function(mandel, len = 4, pix = 400, center = 0, maxItr = 50, radius = 4)
{
  if (! ("Mandelbrot" %in% class(mandel)))
    stop("input should be a /'Mandelbrot/' object")

  plot(mandel)
  print("please click a point:")
  coord = locator(1)
  re = mandel$plotLength * coord$x + Re(mandel$center) - mandel$plotLength / 2
  im = mandel$plotLength * coord$y + Im(mandel$center) - mandel$plotLength / 2
  re = round(re, digits = 4)
  im = round(im, digits = 4)
  return(generateJulia(constant =  re + im * (0+1i), pix = pix, len = len, center = center,
                       maxItr = 50, radius = 4))
}

## Julia Set C Code
#include <R.h>
#include <math.h>


int convItr(double zRe, double zIm, double cRe, double cIm, int maxItr, double radius)
{
   int i = 0;
   double r = 0;
   while (i < maxItr & r <= radius) {
      double zReN = pow(zRe, 2) - pow(zIm, 2) + cRe;
      zIm = 2*zRe*zIm + cIm;
      zRe = zReN;
      r = sqrt( pow(zIm, 2) + pow(zRe, 2) );
      i++;
   }
   return i;
}
```

```
void julia(int * pix, double * centerRe, double * centerIm, double * len, double *cRe, double
*cIm,
        int * mat, int *maxItr, double *radius)
{
   double delta = *len / *pix;
   double re = *centerRe - (*len) / 2;
   for (int i = 0; i < *pix; i++) {
      double im = *centerIm + (*len) / 2;
      for (int j = 0; j < *pix; j++) {
         mat[*pix * i + j] = convItr(re, im, *cRe, *cIm, *maxItr, *radius);
         im = im - delta;
      }
      re = re + delta;
   }
}


void mandelbrot(int * pix, double * centerRe, double * centerIm, double * len,
            int * mat, int * maxItr, double * radius)
{
   double delta = *len / *pix;
   double re = *centerRe - (*len) / 2;
   for (int i = 0; i < *pix; i++) {
      double im = *centerIm + (*len) / 2;
      for (int j = 0; j < *pix; j++) {
         mat[*pix * i + j] = convItr(0, 0, re, im, *maxItr, *radius);
         im = im - delta;
      }
      re = re + delta;
   }
}

## Arrowhead Curve
trans = function(x, y, alpha, a, b){
  x1 = x*cos(alpha)-y*sin(alpha)+a
  y1 = x*sin(alpha)+y*cos(alpha)+b
  return(c(x1, y1))
}
flip = function(x, y, x0, a, b){
  x1 = 2*x0-x +a
  y1 = y +b
  return(c(x1, y1))
}
pin.xy = function(pin){
  n = length(pin)
  x = pin[[1]][[1]]
  y = pin[[1]][[2]]
  for(i in 2:n){
```

```
   x = rbind(x, pin[[i]][[1]])
   y = rbind(y, pin[[i]][[2]])
  }
  return(cbind(x, y))
}



ArrowheadCurve = function(iter){
  a1 =  2*pi/3
  a3 =  -2*pi/3

  x = as.numeric(c(0, 1, 3, 4))
  y = as.numeric(c(0, sqrt(3), sqrt(3), 0))
  xy = cbind(x, y)
  if(iter == 1){
    class(xy) = c("stc", class(xy))
    return(xy)
    break
  }
  for(j in 2:iter){
    ind = seq(1: (length(xy)/2))
    pin1 = lapply(ind, function(i){trans(xy[i, 1], xy[i, 2], a1, 4*2^(j-1), 0)})
    pin2 = lapply(ind, function(i){flip(xy[i, 1], xy[i, 2], 0, 3*2^(j-1), 2^(j-1)*sqrt(3))})
    pin3 = lapply(ind, function(i){trans(xy[i, 1], xy[i, 2], a3, 2^(j-1), 2^(j-1)*sqrt(3))})

    xy1 = pin.xy(pin1)
    xy2 = pin.xy(pin2)
    xy3 = pin.xy(pin3)

    xy = rbind(xy1, xy2, xy3)
    xy = cbind(rev(xy[,1]), rev(xy[ ,2]))
  }
  class(xy) = c("stc", class(xy))
  return(xy)
}

plot.stc = function(xy, ...){
  plot(xy[, 1], xy[, 2], type = "l", axes=FALSE, xlab = "", ylab = "")
}

## Hilbert Curve
# What can we do?
# create H0
H = CreateHCF()
#Hilbert curve after nsteps
H4 = hilbert.curve( H, nsteps)
#GIF of nsteps
```

```
hibert.curve.GIF( H, nsteps)
#S3 method for plot
plot.Hilbert( H )
################################################

# Create HilbertCurve fractal
CreateHCF <-
 # grap = TRUE means I need to grap four arbitrary points in the window for initializing
 # grap = FASLE and type is "classical" means the initialize H matrix is the classical H matrix
 function( grap = TRUE, type = "classical"){
  if( grap ){
   plot.new()
   box()
   print("please click 4 point:")
   H =  locator( 4, type="o" )
   H = matrix( c( H$x, H$y), ncol = 2 )
  }else{
   if( type == "classical" ){
    #classical Hilbert Curve
    H = matrix( c( 1/4, 1/4,
                   1/4, 3/4,
                   3/4, 3/4,
                   3/4, 1/4 ), byrow = TRUE, nrow = 4, ncol = 2)
    colnames( H ) =c( "row", "col" )
   }else{
    H = matrix( rep( 0, 2*4 ), nrow = 4, ncol = 2)
    H[ 1, ] = c( 1/4, 1/4 )
    H[ 4, ] = c( 3/4, 1/4 )
    H[ 2, 1 ] = runif( 1, 0, 1/2 )
    H[ 2, 2 ] = runif( 1, 1/2, 1 )
    H[ 3, 1 ] =  runif( 1, 1/2, 1)
    H[ 3, 2 ] =  runif( 1, 1/2, 1 )
   }
  }

  class( H ) = c( "Hilbert", class(H))
  return( H )
 }



#Move to Next Step
hilbert.curve <-
 function( H, nsteps ){
  for( i in 1:nsteps ){
   H = NextStep( H )
  }
  class( H ) = c( "Hilbert", class(H))
```

```
  return( H )
 }


NextStep <-
 function( H ){
  H0 = H/2

  H_LU = H0 + matrix( c( rep( 0, nrow( H0 ) ) , rep( 1/2, nrow( H0 ))),  ncol = 2 )

  H_RU = H0 %*% matrix( c( -1, 0, 0, 1), nrow = 2, ncol = 2 ) + matrix( c( rep( 1, nrow( H0 )), rep( 1/2, nrow(H0 ) ) ), ncol = 2 )

  H_LD =( H0 - matrix( c( rep( 1/4, nrow( H0 ) ) , rep( 1/4, nrow( H0 )) ),  ncol = 2 )) %*% matrix( c( 0, 1, -1, 0), nrow = 2, ncol = 2 ) + matrix( c( rep( 1/4, nrow( H0 ) ) , rep( 1/4, nrow( H0 ))),  ncol = 2 )

  H_RD =( H0 %*% matrix( c( -1, 0, 0, 1), nrow = 2, ncol = 2 ) + matrix( c( rep( 1/4, nrow( H0 ) ) , rep( -1/4, nrow( H0 ))),  ncol = 2 )) %*% matrix( c( 0, -1, 1, 0), nrow = 2, ncol = 2 ) + matrix( c( rep( 3/4, nrow( H0 ) ) , rep( 1/4, nrow( H0 ))),  ncol = 2 )

  H = rbind( H_LD[ rev(1:nrow( H_LD)), ], H_LU, H_RU[ rev( 1:nrow( H_RU )), ], H_RD )

  return( H )
 }


plot.Hilbert<-
 function( H ){
  plot(  H[ ,1], H[ ,2], type = "l", pch=19, cex=0.5, ylim = c(0,1),  xlim =c(0,1), ylab=character(1),xlab=character(1),axes=FALSE)
  box()
 }

#animation

hibert.curve.GIF<-
 function( H, nsteps){
  library( animation )
  saveGIF( H.img( H, nsteps ) )

 }

H.img <-
 function( H, nsteps ){
  plot(  H[ ,1], H[ ,2], type = "l", pch=19, cex=0.5, ylim = c(0,1),  xlim =c(0,1), ylab=character(1),xlab=character(1),axes=FALSE)
  box()
```

```
   for( i in 1:nsteps ){
     H = hilbert.curve( H, 1)
     plot(     H[  ,1],  H[  ,2],  type  =  "l",  pch=19,  cex=0.5,  ylim  =  c(0,1),     xlim  =c(0,1),
ylab=character(1),xlab=character(1),axes=FALSE)
     box()
   }
  }

## Pattern: for user to decide the initial curve
#xn+1 = yn - sign(xn) | b xn - c |1/2
#yn+1 = a - xn
#
# a=1
# b=4
# c=60

pattern()

pattern <-function(npoint= 1e5,x0=1, y0=1,  a =1, b=4, c=60 ){

  x = c(x0,rep(NA,npoint-1))
  y = c(y0,rep(NA,npoint-1))
  cor<-rep(0, npoint)

  for (i in 2:npoint){
    #iteration link function
    temp = iterfunc( x[ i-1], y[ i-1], a =1, b=4, c=60  )
    x[ i ] = temp[ 1 ]
    y[ i ] = temp[ 2 ]

    #color definition
    cor[i]<-round(sqrt((x[i]-x[i-1])^2+(y[i]-y[i-1])^2),0 )

  }

  n.c<-length(unique(cor))
  cores<-heat.colors(n.c)

  plot( x, y, pch=".", col=cores[cor])
}

#iteration link function
iterfunc = function( x, y, a , b, c){
  nextx = y - sign(x)*sqrt(abs( b*x - c) )
  nexty = a - x
  return( c( nextx, nexty) )
}
```

```
## Koch
#' export functions


kochPattern = function()
{
  pts = getPattern()
  modKochPattern(pts)
} #interacitvely define a pattern



getPattern = function()
{
  plot(c(0,1), c(0,0), xlim=c(0, 1), ylim=c(-0.5,0.5), type = "l")
  pts = data.frame(x = c(0, numeric(49)), y = c(0, numeric(49)) )
  print("click to choose points, press finish to stop.")
  i = 2
  while(1)
  {
    temp = locator(1)
    if (length(temp) == 0) break;
    pts[i,1] = temp$x; pts[i,2] = temp$y
    lines(c(pts$x[i-1], pts$x[i]), c(pts$y[i-1], pts$y[i]))
    i = i + 1
  }
  pts = pts[1:i, ]
  pts[i,1] = 1; pts[i,2] = 0
  pts
}


modKochPattern = function(pts)
{
  pts$y[pts$y <= 0.01 & pts$y >= -0.01] = 0   #those intended to be zero
  l = numeric(nrow(pts) - 2); alpha = numeric(nrow(pts) - 2)
  for (i in 2: (nrow(pts) - 1))
  {
    l[i - 1] = sqrt(pts[i,1]^2 + pts[i,2]^2)  #sqrt(x^2 + y^2) distance from origin to the point
    alpha[i - 1] = atan( pts[i,2] / pts[i,1] )   # the angle between a side and the horizontal axis
  }
  ptn = data.frame( l = l, alpha = alpha)
  ptn
}  #Given a data frame of x, y coordinates of all points of a pattern, modify it to the form that Koch functions can use.
```

```
koch = function(x1, x2, y1, y2, itr)
{
   pts = matrix(nrow = 5, ncol = 2)
   pts[1,] = c(0, 0); pts[5,] = c(1, 0)
   pts[2,] = c(1/3,0); pts[4,] = c(2/3, 0)
   pts[3,] = c(0.5, 1/3*sqrt(3)/2)
   pts = data.frame(pts); names(pts) = c("x", "y")
   ptn = modPattern(pts)
   KochPlus(x1, x2, y1, y2, itr, ptn)
}


kochPlus = function(x1, x2, y1, y2, itr, ptn)
{
  kochPtr = generateKochPlus(x1, x2, y1, y2, itr, ptn)
  kochPtr = data.frame(x = c(x1, kochPtr[,1]), y = c(y1, kochPtr[,2]), stringsAsFactors = FALSE)
  koch = list(curve = kochPtr, iteration = itr)
  class(koch) = c("KochPlusCurve", class(koch))
  return(koch)
}


kochSnowFlake = function(a, itr)
{
  snow = NULL
  snow[[1]] = koch(-a/2, a/2, 0, 0,  itr)
  snow[[2]] = koch(a/2, 0, 0, -sqrt(3)/2*a,  itr)
  snow[[3]] = koch(0, -a/2, -sqrt(3)/2*a, 0, itr)
  class(snow) = c("KochSnowFlake", class(snow))
  snow
}


plot.KochPlusCurve = function(kochplus, ...)
{
   k = kochplus$curve
   lowerLim = min(range(k$x)[1], range(k$y)[1])  #min of min
   upperLim = max(range(k$x)[2], range(k$y)[2])  #max of max
   plot(k$x, k$y, type = "l", xlim = c(lowerLim, upperLim), ylim = c(lowerLim, upperLim), ...)
}


plot.KochSnowFlake = function(snow, ...)
{
  UpperLimX = max(sapply(1:3, function(t) max(snow[[t]]$curve$x)))
  LowerLimX = min(sapply(1:3, function(t) min(snow[[t]]$curve$x)))
  UpperLimY = max(sapply(1:3, function(t) max(snow[[t]]$curve$y)))
```

```
  LowerLimY = min(sapply(1:3, function(t) min(snow[[t]]$curve$y)))

  plot(0, 0, type = "n", xlim = c(LowerLimX, UpperLimX), ylim = c(LowerLimY, UpperLimY), xlab =
"", ylab = "", ...)
  for (i in 1:3)
    lines(snow[[i]]$curve$x, snow[[i]]$curve$y)
}


genPts = function(x1, x2, y1, y2, ptn)
{
  r = sqrt( (x1 - x2)^2 + (y1 - y2)^2 )
  pts = data.frame(x = numeric(nrow(ptn) + 2), y = numeric(nrow(ptn) + 2))
  pts[1, ] = c(x1, y1)

  for (i in 1:nrow(ptn))
  {
    tempx = x1 + ptn$l[i]*(x2 - x1)
    tempy = y1 + ptn$l[i]*(y2 - y1)
    newx = (tempx - x1) * cos(ptn$alpha[i]) - (tempy - y1) * sin(ptn$alpha[i]) + x1
    newy = (tempx - x1) * sin(ptn$alpha[i]) + (tempy - y1) * cos(ptn$alpha[i]) + y1
    pts[i + 1, ] = c(newx, newy)
  }
  pts[ nrow(ptn) + 2, ] = c(x2, y2)
  pts
}  #utilize a pattern in a given line


generateKochPlus = function(x1, x2, y1, y2, itr, ptn)
{
  if (itr < 1)
    return( c(x2, y2))

  else
  {
    pts = genPts(x1, x2, y1, y2, ptn)
    curvePts = NULL
    for ( i in 1: (nrow(pts) - 1))
      curvePts = rbind(curvePts,
                  generateKochPlus(pts[i,1], pts[i+1, 1], pts[i,2], pts[i+1,2],
                              itr - 1, ptn))
  }
  curvePts
}
```