UNIT V

FRACTALS

Fractals and Self similarity – Peano curves – Creating image by iterated functions –Mandelbrot sets – Julia Sets – Random Fractals

A French/American mathematician Dr Benoit Mandelbrot iscovered Fractals. The word fractal was derived from a Latin word fractus which means broken.

- Fractals are very complex pictures generated by a computer from a single formula.
- They are created using iterations. This means one formula is repeated with slightly different values over and over again, taking into account the results from the previous iteration.
- Fractals are used in many areas such as −

Astronomy − For analyzing galaxies, rings of Saturn, etc.

Biology/Chemistry − For depicting bacteria cultures, Chemical reactions, human anatomy, molecules, plants,

Others − For depicting clouds, coastline and borderlines, data compression, diffusion, economy, fractal art, fractal music, landscapes, special effect, etc.

Examples of Fractals

- Clouds
- Grass
- Fire
- Modeling mountains (terrain)
- Coastline
- Branches of a tree
- Surface of a sponge
- Cracks in the pavement
- Designing antennae

1. FRACTALS AND SELF-SIMILARITY

Many of the curves and pictures have a particularly important property called **self-similar**. This means that they appear the same at every scale: No matter how much one enlarges a picture of the curve, it has the same level of detail.

Types: Fractals can also be classified according to their self-similarity

Exactly self-similar:

if a region is enlarged the enlargement looks exactly like the original.

Statistically self-similar:

A **coastline** or a **seashore** is the area where land meets the sea or ocean, or a line that forms the boundary between the land and the ocean .

Successive Refinement of Curves

A complex curve can be fashioned recursively by repeatedly “refining” a simple curve. The simplest example is the Koch curve

Koch Curves:

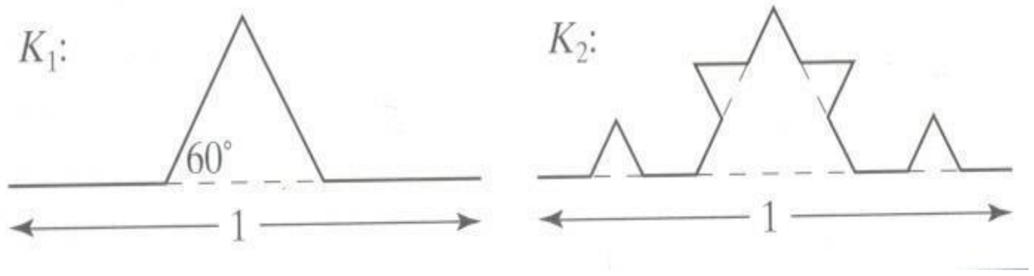
Discovered in 1904 by Helge von Koch Start with straight line of length 1 Recursively:

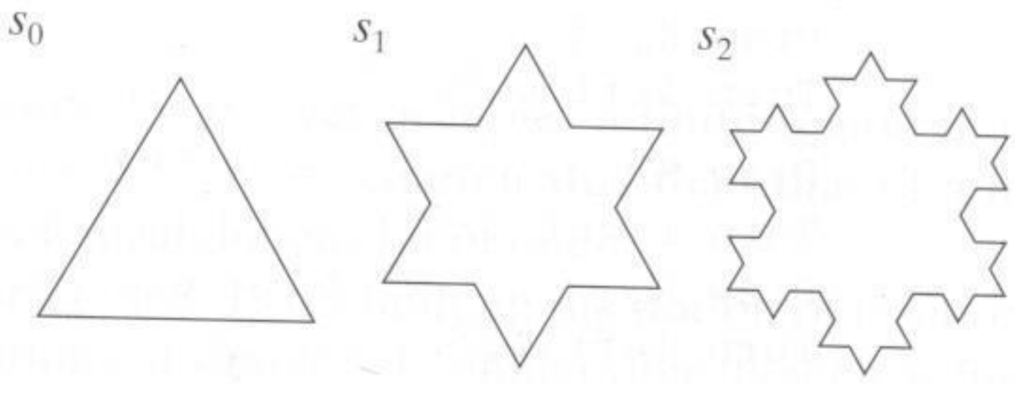
Divide line into 3 equal parts

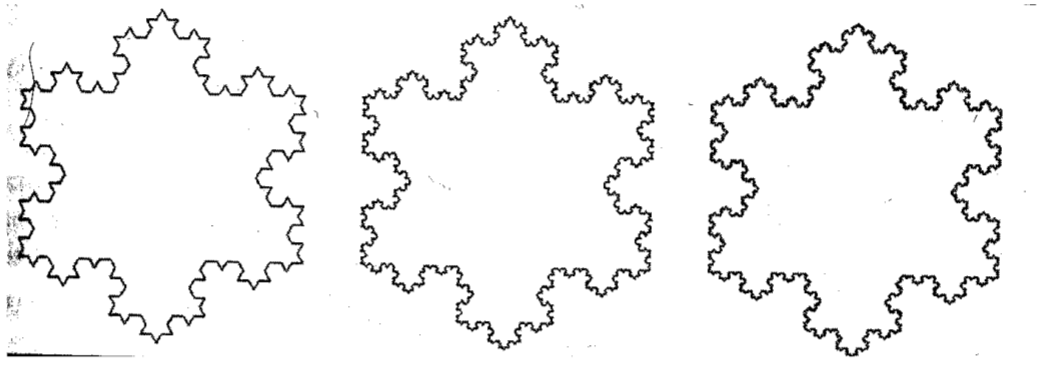
Replace middle section with triangular bump with sides of length 1/3 New length = 4/3

Drawing Koch Curves and Snowflakes

The Koch curves can be viewed in another way: Each generation consists of four versions of the previous generations. For instance K2 consists of four versions of K1 tied end to end with certain angles between them.







- The curve produces an infinitely long line within a region of finite area.

Successive generations of the Koch curve are denoted K0,K1,K2….

The zeroth generation shape K0 is a horizontal line of length unity.

- To create K1 , divide the line K0 into three equal parts and replace the middle section with a triangular bump having sides of length 1/3. The total length of the line is 4/3.
- The second order curve K2, is formed by building a bump on each of the four line segments of K1.

To form Kn+1 from Kn:

- Subdivide each segment of Kn into three equal parts and replace the middle part with a bump in the shape of an equilateral triangle.
- In this process each segment is increased in length by a factor of 4/3, so the total length of the curve is 4/3 larger than that of the previous generation.

Thus Ki has total length of **$(4/3)^i$** , which increases as i increases.
As i tends to infinity, the length of the curve becomes infinite.

Can form Koch snowflake by joining three Koch curves

Perimeter of snowflake grows as:

where Pi is the perimeter of the Ith snowflake iteration

- However, area grows slowly and S = 8/5!!
- Self-similar:
  - zoom in on any portion
  - If n is large enough, shape still same
  - On computer, smallest line segment > pixel spacing

Drawing a Koch Curve

```
Void drawKoch (double dir, double len, int n)
{
//   Koch to order n the line of length len
//   from CP in the direction dir
                                                    // in radians
            double dirRad= 0.0174533 * dir; if (n
            ==0)
               lineRel(len * cos(dirRad), len *
            sin(dirRad)); else {
              n--;                 //reduce the order
              len /=3;             //and the length
              drawKoch(dir, len, n);
              dir +=60;
              drawKoch(dir, len, n);
              dir -=120;
              drawKoch(dir, len, n);
              dir +=60;
              drawKoch(dir, len, n);
             }
  }
```

We call n the order of the curve Kn, and we say the order –n Koch curve consists of four versions of the order (n-1) Koch curve.

To draw K2 we draw a smaller version of K1 , then turn left 60 , draw K1 again, turn right 120 , draw K1 a third time.

For snowflake this routine is performed just three times, with a 120 turn in between.

2. Peano curves

The Peano curves are amongst the first known fractals curves. They were described the first time in 1890 by the italian mathematician Guiseppe Peano.

Construction:

- Start from a octogon drawn inside a square.
- Divide the octogon in nine smaller octogons.
- Link the outer octogons two by two and then those couples to the link all entral octogon.

- Applying the same procedure to the nine small octogon gives rise to the drawing showed here:

Properties :

- **Fractal Dimension**

The fractal dimension is computed using the Hausdorff-esicovitch equation (self-similarity method):

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

Replacing r by three (as each segment is divided by three on each iteration) and N by nine (as the drawing process yields 9 smaller octogons) in the Hausdorff-Besicovitch equation gives:

$$D = \log(9) / \log(3) = 2$$

- **Self-Similarity**

  -Looking at two successive iterations of the drawing process provides graphical evidence that this property is also shared by this curve.

- The Peano curve is a closed curve. The inner space can be filled to increase the contrast with the surrounding background

3. Creating An Image By Means of Iterative Function Systems

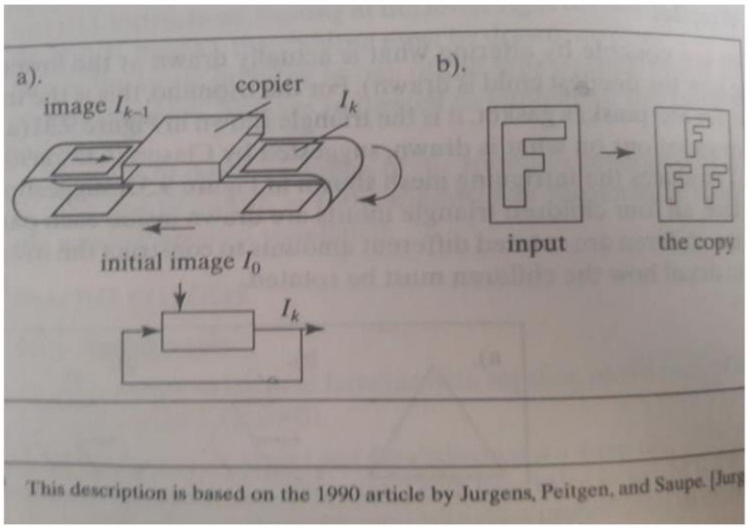
Another way to approach infinity is to apply a transformation to a picture again and again and examine the results.
This technique also provides an another method to create fractal shapes.

An Experimental Copier

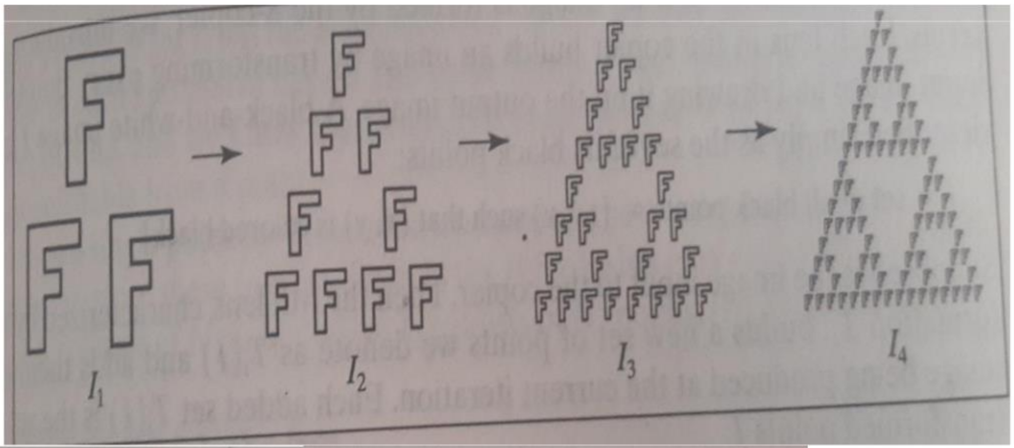
We take an initial image I0 and put it through a special photocopier that produces a new image I1. I1 is not a copy of I0 rather it is a superposition of several reduced versions of I0. We then take I1 and feed it back into the copier again, to produce image I2. This process is repeated , obtaining a sequence of images I0, I1, I2,… called the **orbit of I0**.

In general this copier will have N lenses, each of which perform an affine mapping and then adds its image to the output. The collection of the N affine transformations is called an “iterated function system”.

An iterated function system is a collection of N affine transformations Ti, for i=1,2,…N.



This description is based on the 1990 article by Jurgens, Peitgen, and Saupe. [Jurg



$$M_1 = \begin{pmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad M_2 = \begin{pmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

$$M_3 = \begin{pmatrix} \frac{1}{2} & 0 & \frac{1}{4} \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

Affine Transformation :
T ={m11,m12,m21,m22,m13,m23}
The first four terms contain the elements that performs scalling and rotation.

The final 2 terms perform Translation
Thus the three maps of the s-copiers are
T1={ ½,0,0, ½,0,0}
T2={ ½,0,0, ½,½,0}
T3={ ½,0,0, ½ , ¼,¼}

Underlying Theory of the Copying Process

Each lens in the copier builds an image by transforming every point in the input image and drawing it on the output image. A black and white image I can be described simply as the set of its black points:

I = set of all black points = { (x,y) such that (x,y) is colored black }

I is the input image to the copier. Then the ith lens characterized by transformation Ti, builds a new set of points we denote as Ti(I) and adds them to the image being produced at the current iteration. Each added set Ti(I) is the set of all transformed points I:

Ti(I) = { (x1,y1) | (x1,y1) = Ti(P) for some point P in I }

Upon superposing the three transformed images, we obtain the output image as the union of the outputs from the three lenses:

Output image = T1(I) U T2(I) U T3(I)

The overall mapping from input image to output image as W(.). It maps one set of points – one image – into another and is given by:

W(.)=T1(.) U T2(.) U T3(.)

For instance the copy of the first image I0 is the set W(I0).

Each affine map reduces the size of its image at least slightly, the orbit converge to a unique image called the **attractor** of the IFS. We denote the attractor by the set A, some of its important properties are:

1. The attractor set A is a fixed point of the mapping W(.), which we write as W(A)=A. That is putting A through the copier again produces exactly the same image A.

   The iterates have already converged to the set A, so iterating once more makes no difference.
2. Starting with any input image B and iterating the copying process enough times, we find that the orbit of images always converges to the same A.

   If Ik = $W^{(k)}$(B) is the kth iterate of image B, then as k goes to infinity Ik becomes indistinguishable from the attractor A.

Drawing the kth Iterate

We use graphics to display each of the iterates along the orbit. The initial image I0 can be set, but two choices are particularly suited to the tools developed:

- I0 is a polyline. Then successive iterates are collections of polylines.
- I0 is a single point. Then successive iterates are collections of points.

Using a polyline for I0 has the advantage that you can see how each polyline is reduced in size in each successive iterate. But more memory and time are required to draw each polyline and finally each polyline is so reduced as to be indistinguishable from a point.

Using a single point for I0 causes each iterate to be a set of points, so it is straight forward to store these in a list. Then if IFS consists of N affine maps, the first iterate I1 consists of N points,
image I2 consists of N^2 points, I3 consists of N^3 points, etc.

Copier Operation pseudocode(recursive version)

```
void superCopier( RealPolyArray pts, int k)
{ //Draw kth iterate of input point list pts for the IFS
        int i;                          //reserve space for new list
        RealPolyArray newpts;           //apply each affine
        if(k==0) drawPoints(pts); else
        for(i=1; i<=N; i++) {
                newpts.num= N * pts.num; //the list size grows fast
                for(j=0; j<newpts.num; j++) //transforms the jth point
                    transform(affines[i], pts.pt[j], newpts.pt[j]);
                superCopier(newpts, k – 1);
        }
}
```

- If k=0 it draws the points in the list

  If k>0 it applies each of the affine maps Ti, in turn, to all of the points, creating a new list of points, newpts, and then calls superCopier(newpts, k – 1);

To implement the algorithm we assume that the affine maps are stored in the global array Affine affines[N].

Drawbacks

- Inefficient
- Huge amount of memory is required.

Adding Color

The pictures formed by playing the Chaos Game are bilevel, black dots on a white background. It is easy to extend the method so that it draws gray scale and color images of objects. The image is viewed as a collection of pixels and at each iteration the transformed point lands in one of the pixels. A counter is kept for each pixel and at the completion of the game the number of times each pixel has been visited is converted into a color according to some mapping.

4. THE MANDELBROT SET

Graphics provides a powerful tool for studying a fascinating collection of sets that are the most complicated objects in mathematics.

Julia and Mandelbrot sets arise from a branch of analysis known as iteration theory, which asks what happens when one iterates a function endlessly. Mandelbrot used computer graphics to perform experiments.

Mandelbrot Sets and Iterated Function Systems

A view of the Mandelbrot set is shown in the below figure. It is the black inner portion, which appears to consist of a cardoid along with a number of wartlike circles glued to it.

Its border is complicated and this complexity can be explored by zooming in on a portion of the border and computing a close up view. Each point in the figure is shaded or colored according to the outcome of an experiment run on an IFS.

The Mandelbrot Set

The Iterated function systems for Julia and Mandelbrot sets

The IFS uses the simple function

$f(z) = z^2 + c$-------------------------(1)

where c is some constant. The system produces each output by squaring its input and adding c. We assume that the process begins with the starting value s, so the system generates the sequence of values or orbit

$d1= (s)^2 + c$

$d2= ((s)^2 + c)^2 + c$

$d3= (((s)^2 + c)^2 + c)^2 + c$

$$d4= ((((s)^2 + c)^2 + c)^2 + c)^2 + c \text{ ------------------------------(2)}$$

The orbit depends on two

- ingredients the starting point s
- the given value of c

Given two values of s and c how do points dk along the orbit behaves as k gets larger and larger? Specifically, does the orbit remain finite or explode. Orbits that remain finite lie in their corresponding Julia or Mandelbrot set, whereas those that explode lie outside the set.

When s and c are chosen to be complex numbers , complex arithmetic is used each time the function is applied. The Mandelbrot and Julia sets live in the complex plane – plane of complex numbers.

The IFS works well with both complex and real numbers.

Both s and c are complex numbers and at each iteration we square the previous result and add c. Squaring a complex number z = x + yi yields the new complex number:

$$(x + yi)2 = (x2 – y2) + (2xy)i\text{----------------} \qquad (3)$$

having real part equal to $x^2 – y^2$ and imaginary part equal to 2xy.

Some Notes on the Fixed Points of the System

It is useful to examine the fixed points of the system

f(.) =(.)2 + c . The behavior of the orbits depends on these fixed points that is those complex numbers z that map into themselves, so that $z^2 + c = z$. This gives us the quadratic equation $z^2 – z + c = 0$ and the fixed points of the system are the two solutions of this equation, given by

$$p+, p- = \frac{1}{2} \quad \overline{\frac{1}{4} \quad c} \qquad \text{-------------------------------(4)}$$

If an orbit reaches a fixed point, p its gets trapped there forever. The fixed point can be characterized as **attracting** or **repelling**.

If an orbit flies close to a fixed point p, the next point along the orbit will be forced

- closer to p if p is an attracting fixed point
- farther away from p if p is a repelling a fixed point.

If an orbit gets close to an attracting fixed point, it is sucked into the point. In contrast, a repelling fixed point keeps the orbit away from it.

Defining the Mandelbrot Set

The Mandelbrot set considers different values of c, always using the starting point s =0. For each value of c, the set reports on the nature of the orbit of 0, whose first few values are as follows:

orbit of 0: $0, c, c^2+c, (c^2+c)^2+c, ((c^2+c)^2+c)^2 +c,$……..

For each complex number c, either the orbit is **finite** so that how far along the orbit one goes, the values remain finite or the orbit **explodes** that is the values get larger without limit. The Mandelbrot set denoted by M, contains just those values of c that result in finite orbits:

- The point c is in M if 0 has a finite orbit.

The point c is not in M if the orbit of 0 explodes.

Definition:

The Mandelbrot set M is the set of all complex numbers c that produce a finite orbit of 0.

If c is chosen outside of M, the resulting orbit explodes. If c is chosen just beyond the border of M, the orbit usually thrashes around the plane and goes to infinity.

If the value of c is chosen inside M, the orbit can do a variety of things. For some c's it goes immediately to a fixed point or spirals into such a point.

Computing whether Point c is in the Mandelbrot Set

A routine is needed to determine whether or not a given complex number c lies in M. With a starting point of s=0, the routine must examine the size of the numbers dk along the orbit. As k increases the value of $|d_k|$ either explodes(c is not in M) or does not explode(c is in M).

A theorem from complex analysis states that if $|d_k|$ exceeds the value of 2, then the orbit will explode at some point. The number of iterations $|d_k|$ takes to exceed 2 is called the **dwell** of the orbit.

But if c lies in M, the orbit has an infinite dwell and we can't know this without it iterating forever. We set an upper limit Num on the maximum number of iterations we are willing to wait for.

A typical value is Num = 100. If $|d_k|$ has not exceeded 2 after Num iterates, we assume that it will never and we conclude that c is in M. The orbits for values of c just outside the boundary of M have a large dwell and if their dwell exceeds Num, we wrongly decide that they lie inside M. A drawing based on too small value of Num will show a Mandelbrot set that is slightly too large.

dwell() routine int dwell (double cx,

```
        double cy)
        { // return true dwell or Num, whichever is smaller
                #define Num 100 // increase this for better pictures
                double tmp, dx=cx, dy=cy, fsq=cx *cx + cy * cy;
                for(int count=0; count<=Num && fsq <=4; count++)
                {
                  tmp = dx;                    //save old real part
                  dx = dx * dx – dy * dy +cx;            //new real part
                  dy = 2.0 * tmp * dy + cy;              //new imaginary part
                  fsq = dx * dx + dy * dy;
                }
               return count;                   // number of iterations used
          }
```

For a given value of c = cx + cyi, the routine returns the number of iterations required for $|d_k|$ to exceed 2.

At each iteration, the current dk resides in the pair (dx,dy) which is squared using eq(3) and then added to (cx,cy) to form the next d value. The value $d\ k|^2$ is kept in fsq and compared with 4. The dwell() function plays a key role in drawing the Mandelbrot set.

Drawing the Mandelbrot Set

Pseudocode for drawing a region of the Mandelbrot set

```
for(j=0; j<rows; j++)
   for(i=0; i<cols; i++)
       {
          find the corresponding c value in equation
          estimate the dwell of the orbit
          find Color determined by estimated
          dwell setPixel( j , k, Color);
       }
```

A practical problem is to study close up views of the Mandelbrot set, numbers must be stored and manipulated with great precision.

Also when working close to the boundary of the set , you should use a larger value of Num. The calculation times for each image will increase as you zoom in on a region of the boundary of M. But images of modest size can easily be created on a microcomputer in a reasonable amount of time.

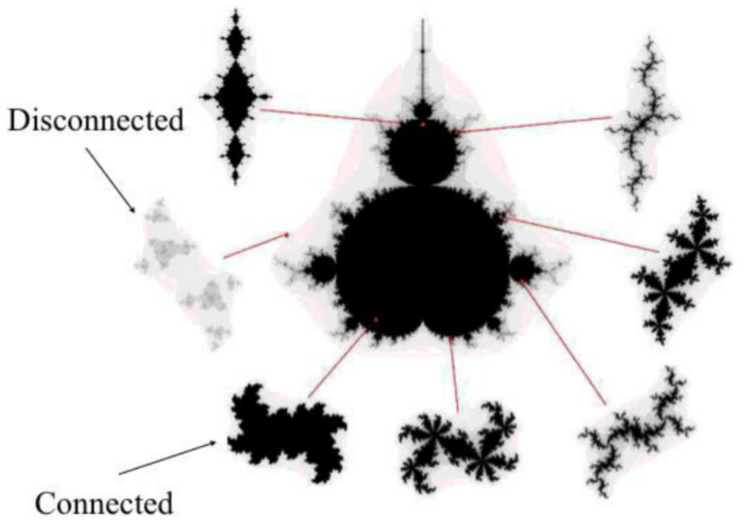
5. JULIA SETS

The Julia Set(s) are a slightly different than the Mandelbrot Set as I will explain. For every POINT in the complex plane, there exists a set called the Julia Set each of which has a different shape and a different dynamic associated with it. In other words, every point in the complex plane generates a different Julia Set "curve" or image.

Also, it has been shown that the Julia Sets generated from the points on the INSIDE of the Mandelbrot Set form Connected Sets, and the Julia Sets that are generated from the points on the OUTSIDE of the M-Set form disconnected sets also known as Cantor dusts.

- Julia sets are extremely complicated sets of points in the complex plane.
- There is a different Julia set, denoted Jc for each value of c. A closely related variation is the filled-in Julia set,
- denoted by Kc, which is easier to define.

The Julia Set (s)



The Filled-In Julia Set Kc

- The filled-in Julia set at c, Kc, is the set of all starting points whose orbits are finite.
- When studying Kc, one chooses a single value for c and considers different starting points.
- Kc should be always symmetrical about the origin, since the orbits of s and –s become identical after one iteration.

Drawing Filled-in Julia Sets

A starting point s is in Kc, depending on whether its orbit is finite or explodes, the process of drawing a filled-in Julia set is almost similar to Mandelbrot set. We choose a window in the complex plane and associate pixels with points in the window. The pixels correspond to different values of the starting point s. A single value of c is chosen and then the orbit for each pixel position is examined to see if it explodes and if so, how quickly does it explodes.

Pseudocode for drawing a region of the Filled-in Julia set

```
for(j=0; j<rows; j++)
    for(i=0; i<cols; i++)
        {
           find the corresponding s value in equation
           (5) estimate the dwell of the orbit
           find Color determined by estimated
           dwell setPixel( j , k, Color);
        }
```

The dwell() must be passed to the starting point s as well as c. Making a high-resolution image of a Kc requires a great deal of computer time, since a complex calculation is associated with every pixel.

Notes on Fixed Points and Basins of Attraction

If an orbit starts close enough to an attracting fixed point, it is sucked into that point. If it starts too far away, it explodes. The set of points that are sucked in forms a so called **basin of attraction** for the fixed point p. The set is the filled-in Julia set Kc. The fixed point which lies inside the circle |z|= ½ is the attracting point.All points inside Kc, have orbits that explode. All points inside Kc, have orbits that spiral or plunge into the attracting fixed point.

If the starting point is inside Kc, then all of the points on the orbit must also be inside Kc and they produce a finite orbit. The repelling fixed point is on the boundary of Kc.

Kc for Two Simple Cases The set Kc is simple for two values of c:

1. **c=0:** Starting at any point s, the orbit is simply s, s2,s4,…….,s2k,…, so the orbit spirals into 0 if |s|<1 and explodes if |s|>1.

Thus K0 is the set of all complex numbers lying inside the unit circle, the

circle of radius 1 centered at the origin.

2. **c = -2:** in this case it turns out that the filled-in Julia set consists of all points lying on the real axis between -2 and 2.

For all other values of c, the set Kc, is complex. It has been shown that each Kc is one of the two types:

- Kc is connected or
- Kc is a Cantor set

A theoretical result is that Kc is connected for precisely those values of c that lie in the Mandelbrot set.

Julia Set Jc is for any given value of c; it is the boundary of Kc. Kc is the set of all starting points that have finite orbits and every point outside Kc has an exploding orbit. We say that the points just along the boundary of Kc and "on the fence". Inside the boundary all orbits remain finite; just outside it, all orbits goes to infinity.

If the process started instead at f(s), the image of s, then the two orbits would be:

s, f(s), f2(s), f3(s),….   (orbit of s)

or

f(s), f2(s), f3(s), f4(s),….   (orbit of f(s))

which have the same value forever. If the orbit of s is finite, then so is the orbit of its image f(s). All of the points in the orbit , if considered as starting points on their own, have orbits with thew same behavior: They all are finite or they all explode.

Any starting point whose orbit passes through s has the same behavior as the orbit that start at s: The two orbits are identical forever.

The point "**just before**" s in the sequence is called the **preimage** of s and is the inverse of the function f(.) = $(.)^2$ + c.

Drawing the Julia set Jc

To draw Jc we need to find a point and place a dot at all of the point's preimages. Therea re two problems with this method:

1. finding a point in Jc
2. keeping track of all the preimages

An approach known as the backward-iteration method overcomes these obstacles and produces good result. The idea is simple: Choose some point z in the complex plane. The point may or may not be in Jc. Now iterate in backward direction: at each iteration choose one of the two square roots randomly, to produce a new z value

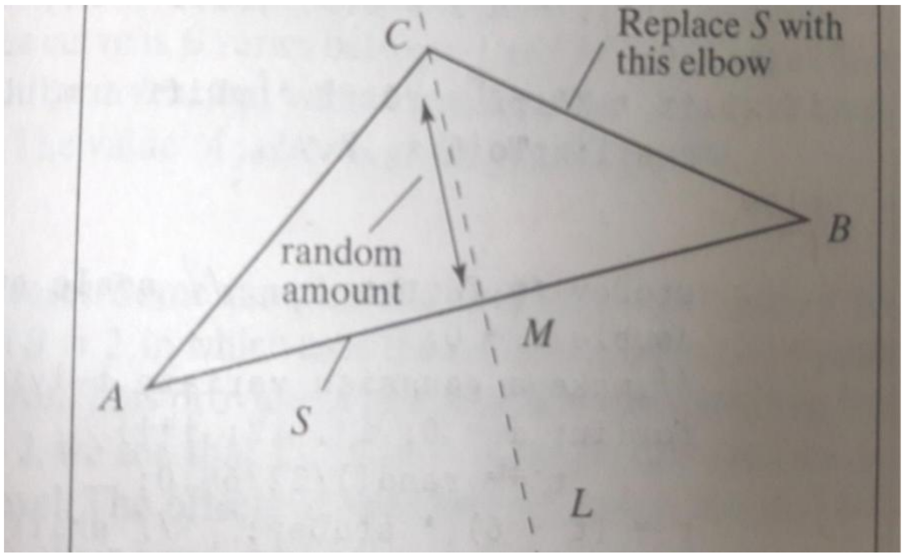
5. RANDOM FRACTALS

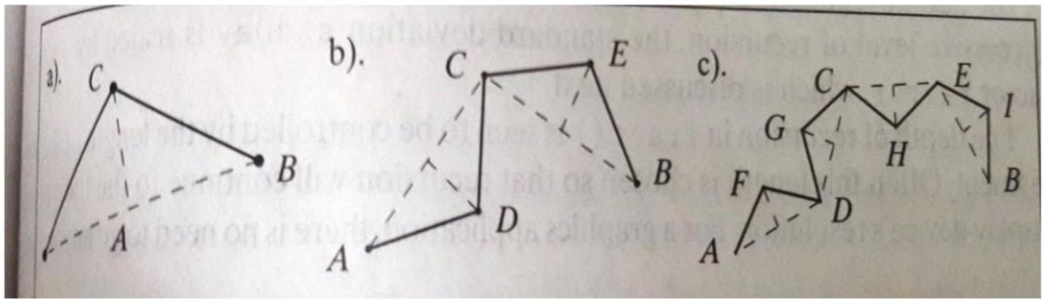
Fractal is the term associated with randomly generated curves and surfaces that exhibit a degree of self-similarity. These curves are used to provide "naturalistic" shapes for representing objects such as coastlines, rugged mountains, grass and fire.

Fractalizing a Segment

The simplest random fractal is formed by recursively roughening or fractalizing a line segment. At each step, each line segment is replaced with a "random elbow".

The figure shows this process applied to the line segment S having endpoints A and B. S is replaced by the two segments from A to C and from C to B. For a fractal curve, point C is randomly chosen along the perpendicular bisector L of S. The elbow lies randomly on one or the other side of the "parent" segment AB.





- Three stages are required in the fractalization of a segment.
- In the first stage, the midpoint of AB is perturbed to form point C.
- In the second stage , each of the two segment has its midpoints perturbed to form points D and E.
- In the final stage, the new points F…..I are added.

To perform afractalization in a program:

Line L passes through the midpoint M of segment S and is perpendicular to it. Any point C along L has the parametric form:

$$\mathbf{C(t) = M + (B\text{-}A)^{\perp} t}$$

for some values of t, where the midpoint M= (A+B)/2.

The distance of C from M is |B-A||t|, which is proportional to both t and the length of S. So to produce a point C on the random elbow, we let t be computed randomly. If t is positive, the elbow lies to one side of AB; if t is negative it lies to the other side.

For most fractal curves, t is modeled as a Gaussian random variable with a zero mean and some standard deviation. Using a mean of zero causes, with equal probability, the elbow to lie above or below the parent segment.

Fractalizing a Line segment

```
void fract(Point2 A, Point2 B, double stdDev)
// generate a fractal curve from A to B
 double xDiff = A.x – B.x, yDiff=
 A.y –B.y;
Point2 C;
if(xDiff * XDiff + YDiff * yDiff < minLenSq)
cvs.lintTo(B.x, B.y);
else
{
stdDev *=factor;    //scale stdDev by factor
double t=0;
// make a gaussian variate t lying between 0
 and 12.0 for(int i=0; I< 12; i++)
t+= rand()/32768.0;
t= (t-6) * stdDev; //shift the mean to 0 and
sc C.x = 0.5 *(A.x +B.x) – t * (B.y –
A.y); C.y = 0.5 *(A.y +B.y) – t * (B.x –
A.x);
          fract(A, C, stdDev);
           fract(C, B, stdDev);
}
```

The routine fract() generates curves that approximate actual fractals. The routine recursively replaces each segment in a random elbow with a smaller random elbow.

The stopping criteria used is: When the length of the segment is small enough, the segment is drawn using cvs.lineTo(), where cvs is a Canvas object.

The variable t is made to be approximately Gaussian in its distribution by summing together 12 uniformly distributed random values lying between 0 and 1.

The result has a mean value of 6 and a variance of 1. The mean value is then shifted to 0 and the variance is scaled as necessary.The depth of recursion in fract() is controlled by the length of the line segment.

Controlling the Spectral Density of the Fractal Curve:

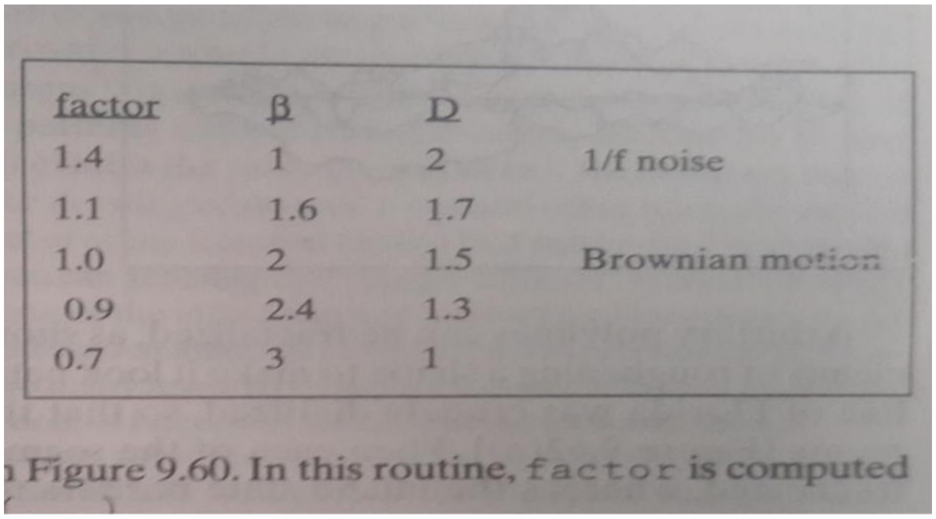
The fractal dimension of such processes is:

The fractal curve generated using the above
code has a “power spectral density” given by

$$S(f)= 1/f^{\beta}$$

Where β the power of the noise process is the parameter the user can set to control the jaggedness of the fractal noise.
When β is 2, the process is known as Brownian motion and when β is 1, the process is called “1/f noise”. 1/f noise is self similar and is shown to be a good model for physical process such as clouds.

In the routine fract(), the scaling factor factor by which the standard deviation is scaled at each level based on the exponent β of the fractal curve. Values larger than 2 leads to smoother curves and values smaller than 2 leads to more jagged curves. The value of factor is given by: factor = 2 (1 – β/2)
The factor decreases as β increases.



factor	β	D	
1.4	1	2	1/f noise
1.1	1.6	1.7	
1.0	2	1.5	Brownian motion
0.9	2.4	1.3	
0.7	3	1	

Figure 9.60. In this routine, factor is computed

Drawing a fractal curve(pseudocode)

```
double MinLenSq, factor; //global variables

void drawFractal (Point2 A, Point2 B) {

double beta, StdDev;

User inputs beta, MinLenSq and the the initial StdDev

factor = pow(2.0, (1.0 – beta)/ 2.0); cvs.moveTo(A);

fract(A, B, StdDev); }
```