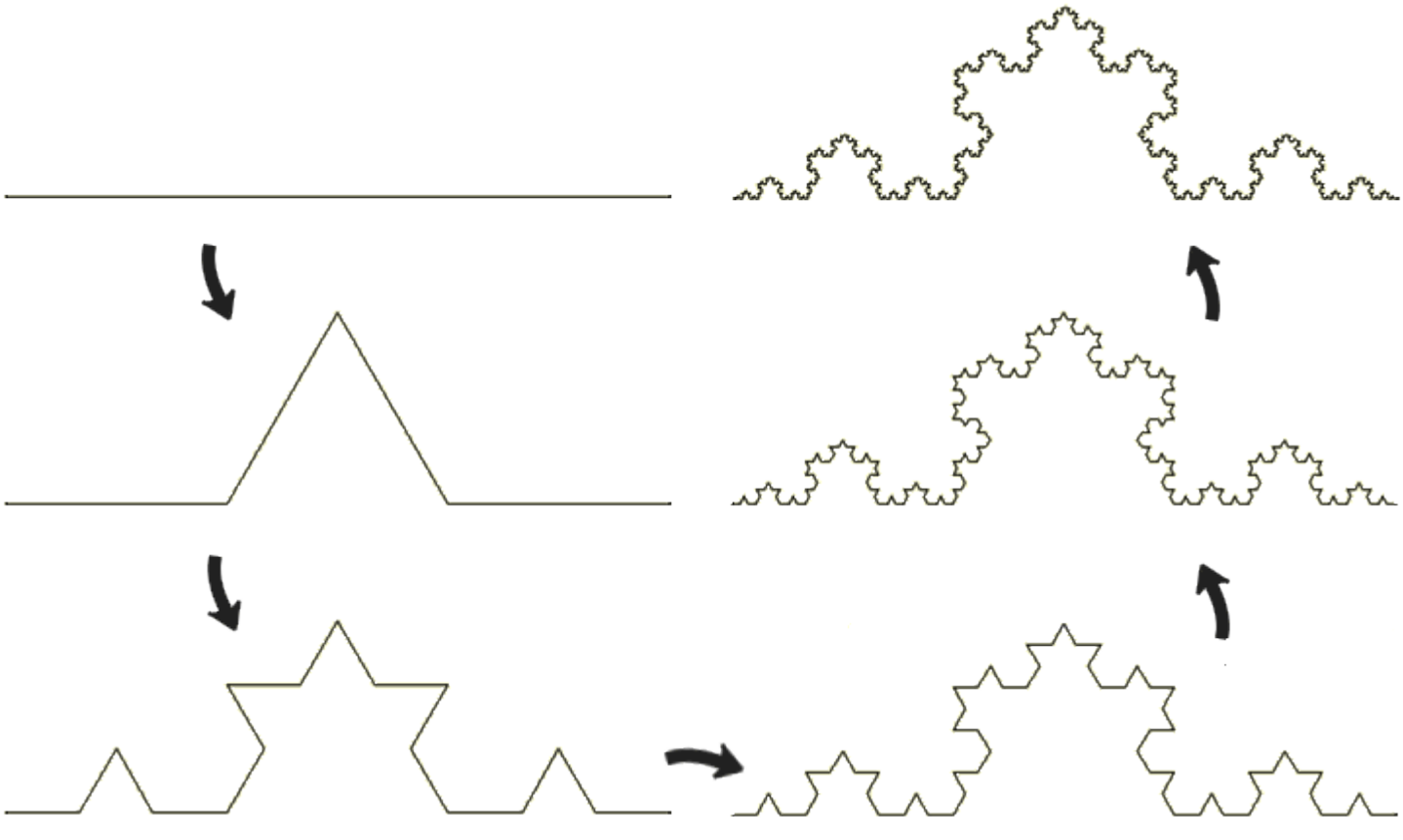
神奇的分形艺术（一）：无限长的曲线可能围住一块有限的面积

很多东西都是吹神了的，其中麦田圈之谜相当引人注目。上个世纪里人们时不时能听见某个农民早晨醒了到麦田地一看立马吓得屁滚尿流的故事。上面这幅图就是97年在英国 Silbury 山上发现的麦田圈，看上去大致上是一个雪花形状。你或许会觉得这个图形很好看。看了下面的文字后，你会发现这个图形远远不是“好看”可以概括的，它的背后还有很多东西。

在说明什么是分形艺术前，我们先按照下面的方法构造一个图形。看下图，首先画一个线段，然后把它平分成三段，去掉中间那一段并用两条等长的线段代替。这样，原来的一条线段就变成了四条小的线段。用相同的方法把每一条小的线段的中间三分之一替换为等边三角形的两边，得到了16条更小的线段。然后继续对16条线段进行相同的操作，并无限地迭代下去。下图是这个图形前五次迭代的过程，可以看到这样的分辨率下已经不能显示出第五次迭代后图形的所有细节了。这样的图形可以用 Logo 语言很轻松地画出来。



你可能注意到一个有趣的事实：整个线条的长度每一次都变成了原来的4/3。如果最初的线段长为一个单位，那么

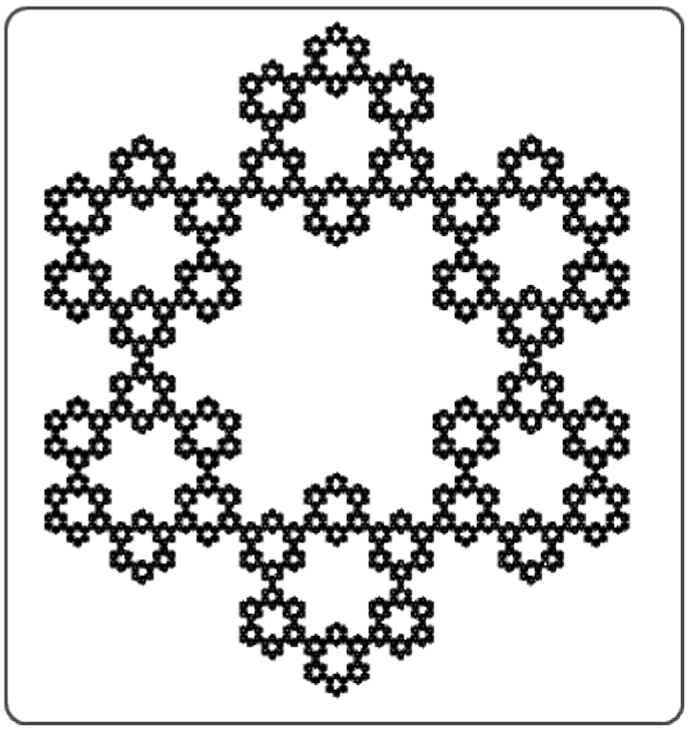
第一次操作后总长度变成了4/3，第二次操作后总长增加到16/9，第 n 次操作后长度为(4/3)^n。毫无疑问，操作无限进行下去，这条曲线将达到无限长。难以置信的是这条无限长的曲线却“始终只有那么大”。

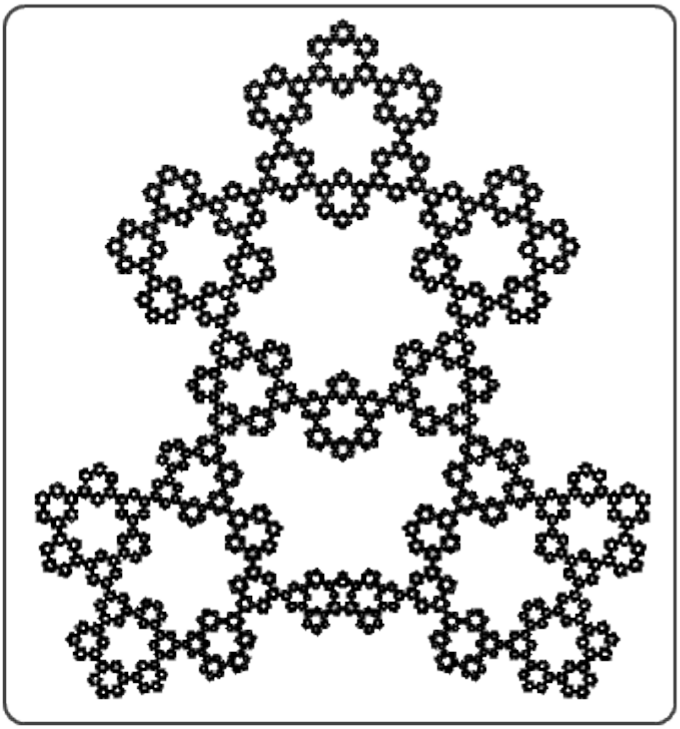
当把三条这样的曲线头尾相接组成一个封闭图形时，有趣的事情发生了。这个雪花一样的图形有着无限长的边界，但是它的总面积却是有限的。换句话说，无限长的曲线围住了一块有限的面积。有人可能会问为什么面积是有限的。虽然从上面的图上看结论很显然，但这里我们还是要给出一个简单的证明。三条曲线中每一条的第 n 次迭代前有4^(n-1)个长为(1/3)^(n-1)的线段，迭代后多出的面积为4^(n-1)个边长为(1/3)^n 的等边三角形。把4^(n-1)扩大到4^n，再把所有边长为(1/3)^n 的等边三角形扩大为同样边长的正方形，总面积仍是有限的，因为无穷级数Σ4^n/9^n 显然收敛。这个神奇的雪花图形叫做 Koch 雪花，其中那条无限长的曲线就叫做 Koch 曲线。他是由瑞典数学家 Helge von Koch 最先提出来的。本文最开头提到的麦田圈图形显然是想描绘 Koch 雪花。

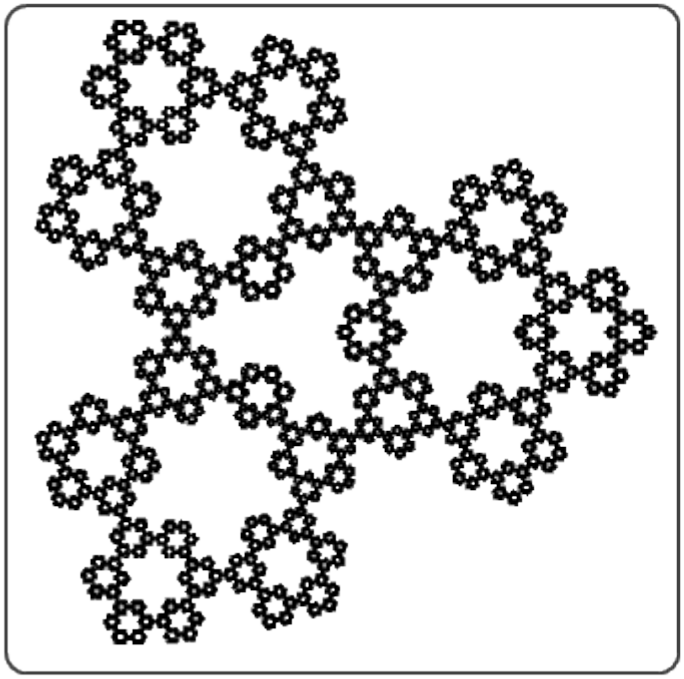
分形这一课题提出的时间比较晚。Koch 曲线于1904年提出，是最早提出的分形图形之一。我们仔细观察一下这条特别的曲线。它有一个很强的特点：你可以把它分成若干部分，每一个部分都和原来一样（只是大小不同）。这样的图形叫做“自相似”图形(self-similar)，它是分形图形(fractal)最主要的特征。自相似往往都和递归、无穷之类的东西联系在一起。比如，自相似图形往往是用递归法构造出来的，可以无限地分解下去。一条 Koch 曲线中包含有无数大小不同的 Koch 曲线。你可以对这条曲线的尖端部分不断放大，但你所看到的始终和最开始一样。它的复杂性不随尺度减小而消失。另外值得一提的是，这条曲线是一条连续的，但处处不光滑（不可微）的曲线。曲线上的任何一个点都是尖点。

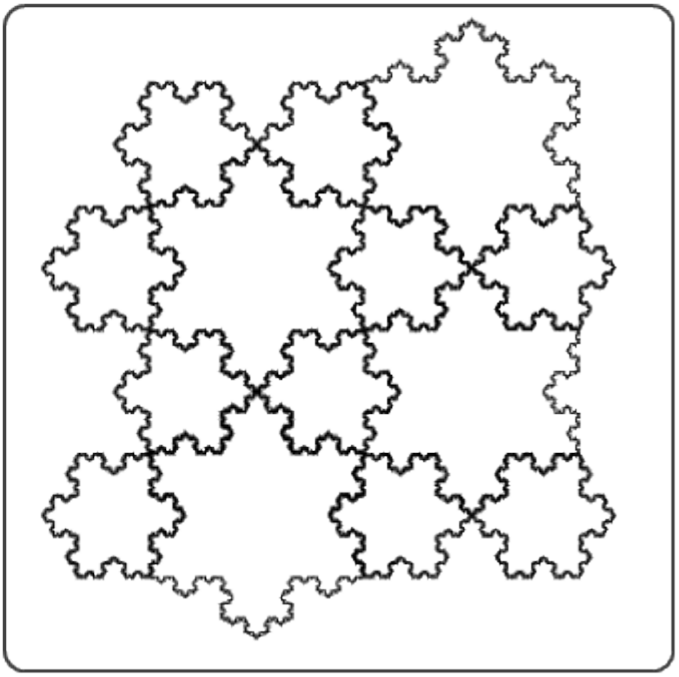
分形图形有一种特殊的计算维度的方法。我们可以看到，在有限空间内就可以达到无限长的分形曲线似乎已经超越了一维的境界，但说它是二维图形又还不够。Hausdorff 维度就是专门用来对付这种分形图形的。简单地说，Hausdorff 维度描述分形图形中整个图形的大小与一维大小的关系。比如，正方形是一个分形图形，因为它可以分成四个一模一样的小正方形，每一个小正方形的边长都是原来的1/2。当然，你也可以把正方形分成9个边长为1/3的小正方形。事实上，一个正方形可以分割为 a^2个边长为1/a 的小正方形。那个指数2就是正方形的维度。矩形、三角形都是一样，给你 a^2个同样的形状才能拼成一个边长为 a 倍的相似形，因此它们都是二维的。我们把这里的“边长”理解为一维上的长度，那个1/a 则是两个相似形的相似比。如果一个自相似形包含自身 N 份，每一份的一维大小都是原来的1/s，则这个相似形的 Hausdorff 维度为 log(N)/log(s)。一个立方体可以分成8份，每一份的一维长度都是原来的一半，因此立方体的维度为 log(8)/log(2)=3。同样地，一个 Koch 曲线包含四个小 Koch 曲线，大小两个 Koch 曲线的相似比为1/3，因此 Koch 曲线的 Hausdorff 维度为 log(4)/log(3)。它约等于1.26，是一个介于1和2之间的实数。

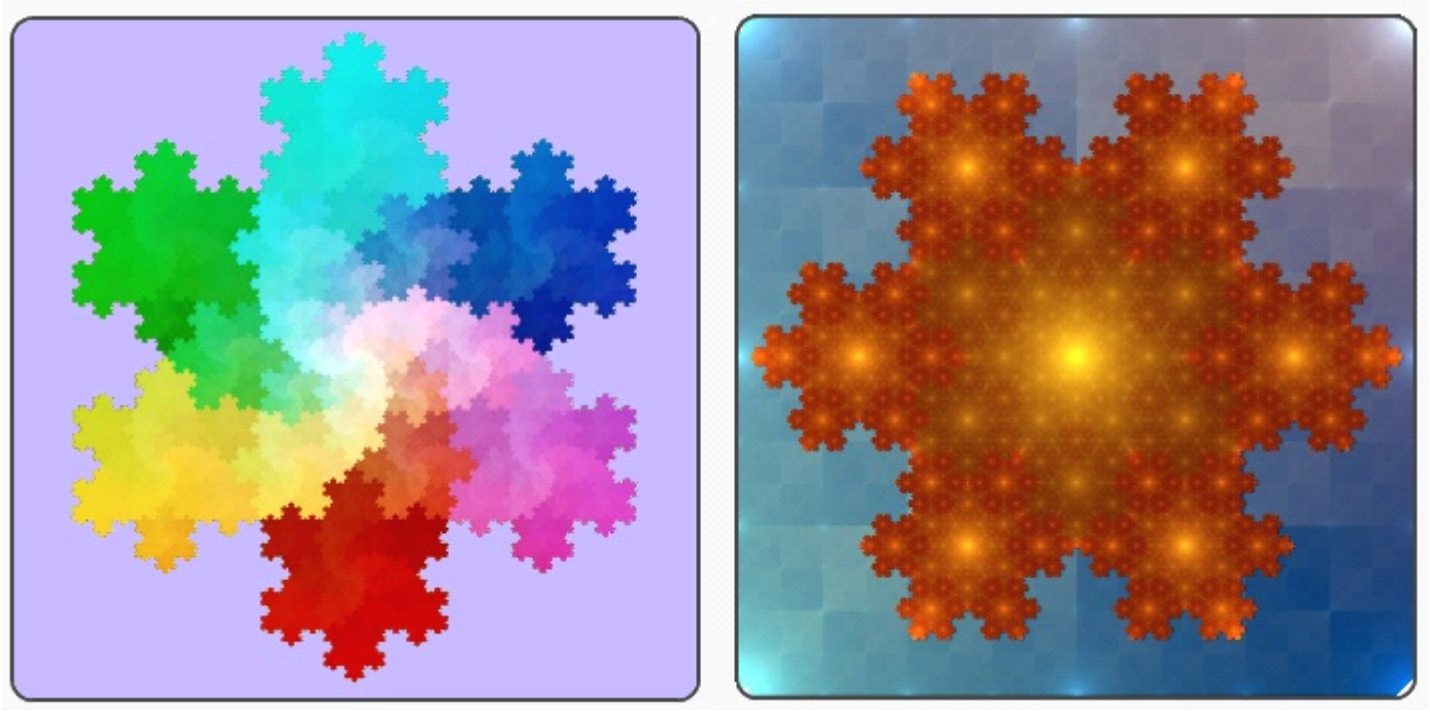
我们常说分形图形是一门艺术。把不同大小的 Koch 雪花拼接起来可以得到很多美丽的图形。如果有 MM 看了前面的文字一句也不懂，下面这些图片或许会让你眼前一亮。



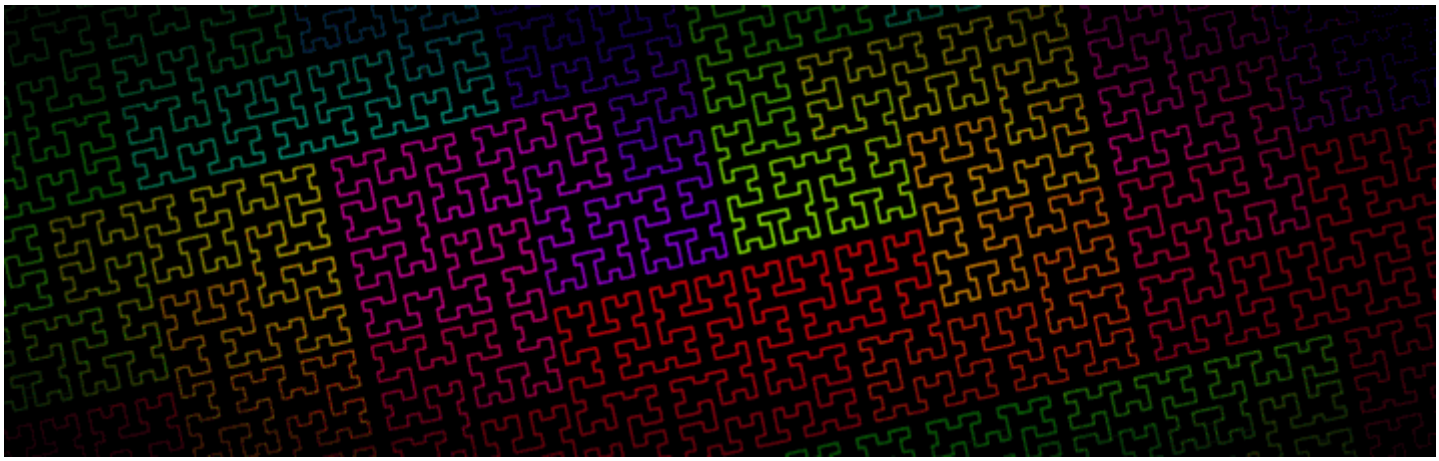








神奇的分形艺术（二）：一条连续的曲线可以填满整个平面



虽然有些东西似乎是显然的，但一个完整的定义仍然很有必要。比如，大多数人并不知道函数的连续性是怎么定义的，虽然大家一直在用。有人可能会说，函数是不是连续的一看就知道了嘛，需要定义么。事实上，如果没有严格的定义，你很难把下面两个问题说清楚。

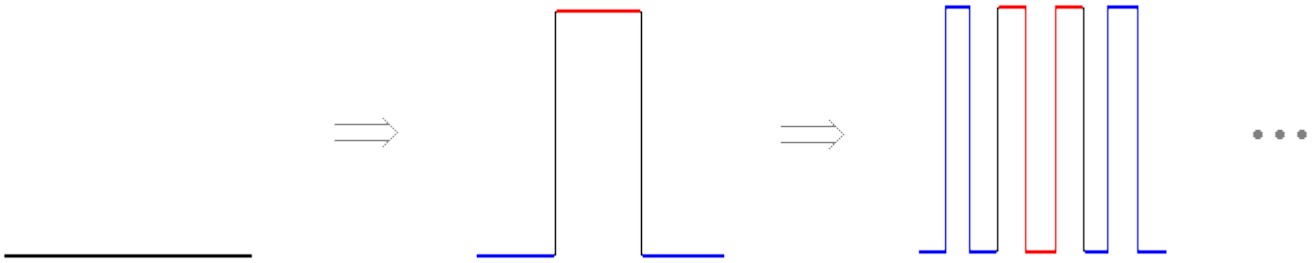
你知道吗，除了常函数之外还存在其它没有最小正周期的周期函数。考虑一个这样的函数：它的定义域为全体实数，当 x 为有理数时 f(x)=1，当 x 为无理数时 f(x)=0。显然，任何有理数都是这个函数的一个最小正周期，因为一个有理数加有理数还是有理数，而一个无理数加有理数仍然是无理数。因此，该函数的最小正周期可以任意小。如果非要画出它的图象，大致看上去就是两根直线。请问这个函数是连续函数吗？如果把这个函数改一下，当 x 为无理数时 f(x)=0，当 x 为有理数时 f(x)=x，那新的函数是连续函数吗？

Cauchy 定义专门用来解决这一类问题，它严格地定义了函数的连续性。Cauchy 定义是说，函数 f 在 x=c 处连续当且仅当对于一个任意小的正数ε，你总能找到一个正数δ使得对于定义域上的所有满足 c-δ< x <c+δ的 x 都有 f(c)-ε<f(x)<f(c)+ε。直观地说，如果函数上有一点 P，对于任意小的ε，P 点左右一定范围内的点与 P 的纵坐标之差均小于ε，那么函数在 P 点处连续。这样就保证了 P 点两旁的点与 P 无限接近，也就是我们常说的“连续”。这又被称作为 Epsilon-Delta 定义，可以写成“ε-δ定义”。

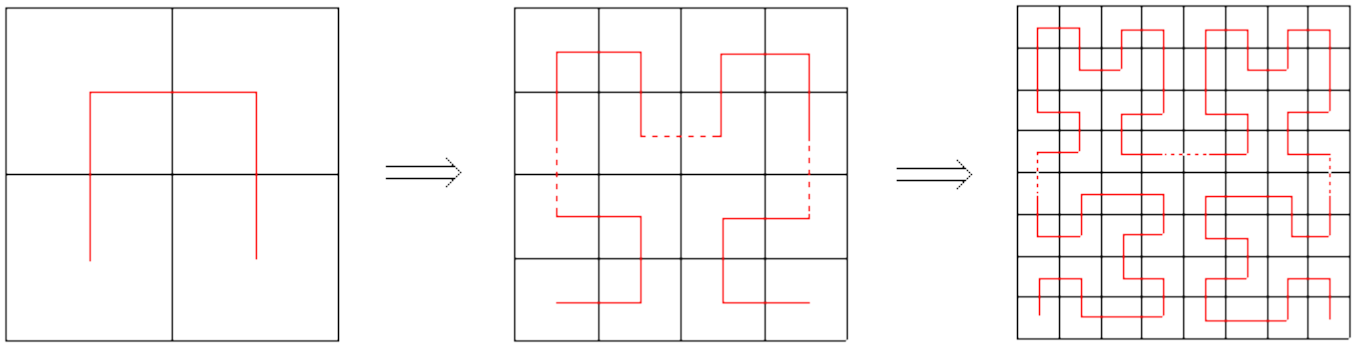
有了 Cauchy 定义，回过头来看前面的问题，我们可以推出：第一个函数在任何一点都不连续，因为当 ε< 1时，δ范围内总存在至少一个点跳出了ε的范围；第二个函数只在 x=0处是连续的，因为此时不管ε是多少，只需要δ比ε小一点就可以满足ε-δ定义了。

在拓扑学中，也有类似于ε-δ的连续性定义。假如一个函数 f(t)对应空间中的点，对于任意小的正数ε，总能找到一

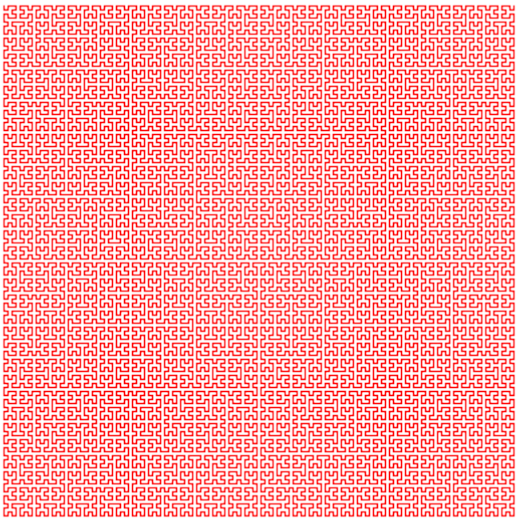
个δ使得定义域(t-δ,t+δ)对应的所有点与 f(t)的距离都不超过ε，那么我们就说 f(t)所对应的曲线在点 f(t)处连续。



回到我们的话题，如何构造一条曲线使得它可以填满整个平面。在这里我们仅仅说明存在一条填满单位正方形的曲线就够了，因为将此单位正方形平铺在平面上就可以得到填满整个平面的曲线。大多数人可能会想到下面这种构造方法：先画一条单位长的曲线，然后把它变成一个几字形，接着把每一条水平的小横线段变成一个几字形，然后不断迭代下去，最后得到的图形一定可以填满整个单位正方形。我们甚至可以递归地定义出一个描述此图形的函数：将定义域平均分成五份，第二和第四份对应两条竖直线段上的点，并继续对剩下的三个区间重复进行这种操作。这个函数虽然分布得有些“不均匀”，但它确实是一个合法的函数。最后的图形显然可以填充一个正方形，但它是不是一条曲线我们还不知道呢。稍作分析你会发现这条“曲线”根本不符合前面所说的ε-δ定义，考虑任何一个可以无限细分的地方（比如 x=1/2处），只要ε<1/2，δ再小其范围内也有一条竖线捅破ε的界线。这就好像当 n 趋于无穷时 sin(nx)根本不是一条确定的曲线一样，因为某个特定的函数值根本不能汇聚到一点。考虑到这一点，我们能想到的很多可以填满平面的“曲线”都不是真正意义上的连续曲线。为了避免这样的情况出现，这条曲线必须“先把自己周围填满再延伸出去”，而填满自己周围前又必须先填满“更小规模的周围”。这让我们联想到分形图形。



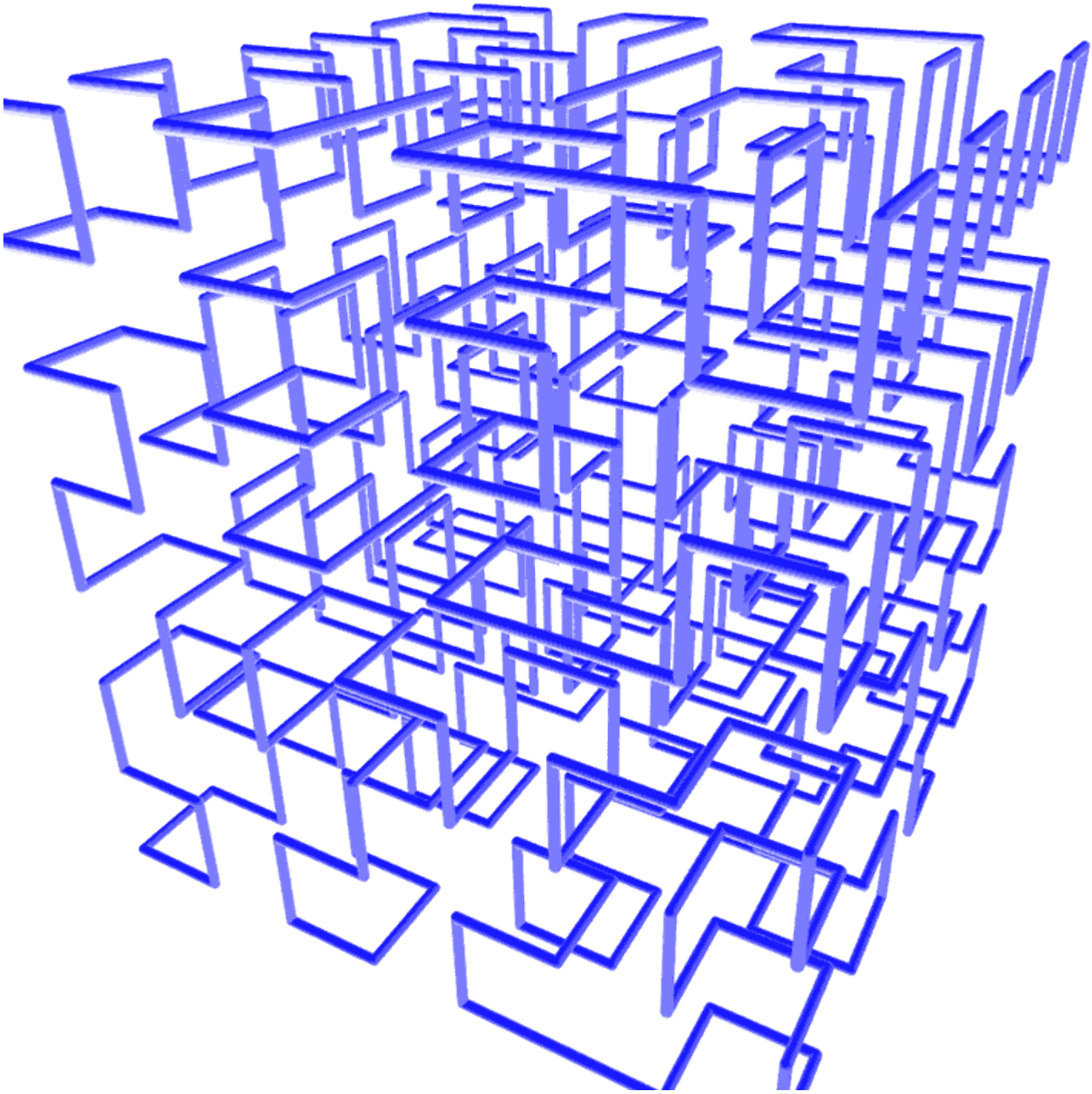
德国数学家 David Hilbert 发现了这样一种可以填满整个单位正方形的分形曲线，他称它为 Hilbert 曲线。我们来看一看这条曲线是怎么构造出来的。首先，我们把一个正方形分割为 4 个小正方形，然后从左下角的那个小正方形开始，画一条线经过所有小正方形，最后到达右下角。现在，我们把这个正方形分成 16 个小正方形，目标同样是从左下角出发遍历所有的格子最后到达右下角。而在这之前我们已经得到了一个 2x2 方格的遍历方法，我们正好可以用它。把两个 2x2 的格子原封不动地放在上面两排，右旋 90 度放在左下，左旋 90 度放在右下，然后再补三条线段把它们连起来。现在我们得到了一种从左下到右下遍历 4x4 方格的方法，而这又可以用于更大规模的图形中。用刚才的方法把四个 4x4 的方格放到 8x8 的方格中，我们就得到了一条经过所有 64 个小方格的曲线。不断地这样做下去，无限多次地迭代后，每个方格都变得无穷小，最后的图形显然经过了方格上所有的点，它就是我们所说的 Hilbert 曲线。下图是一个迭代了 n 多次后的图形，大致上反映出 Hilbert 曲线的样子。



根据上面这种方法，我们可以构造出函数 f(t)使它能映射到单位正方形中的所有点。Hilbert 曲线首先经过单位正方形左下 1/4 的所有点，然后顺势北上，东征到右上角，最后到达东南方的 1/4 正方形，其中的每一个阶段都是一个边长缩小了一半的“小 Hilbert 曲线”。函数 f(t)也如此定义：[0,1/4]对应左下角的小正方形中所有的点，[1/4,1/2]就对应左上角，依此类推。每个区间继续划分为四份，依次对应面积为 1/16 的正方形，并无限制地这么细分下去。注意这里的定义域划分都是闭区间的形式，这并不会发生冲突，因为所有 m/4^n 处的点都是两个小 Hilbert 曲线的“交接处”。比如那个 f(1/4)点就是左上左下两块 1/4 正方形共有的，即单位正方形正左边的那一点。这个函数是一条根正苗红的连续曲线，完全符合ε-δ定义，因为 f(t-δ)和 f(t+δ)显然都在 f(t)的周围。

Hilbert 曲线是一条经典的分形曲线。它违背了很多常理。比如，把 Hilbert 曲线平铺在整个平面上，它就成了一条填满整个平面的曲线。两条 Hilbert 曲线对接可以形成一个封闭曲线，而这个封闭曲线竟然没有内部空间。回想我们上次介绍的 Hausdorff 维度，你会发现这条曲线是二维的，因为它包含自身 4 份，每一份的一维大小都是原来的一半，因此维度等于 log(4)/log(2)。这再一次说明了它可以填满整个平面。

Hilbert 曲线的价值在于建立一维空间与二维空间一一对应的关系。Hilbert 曲线可以看作是一个一维空间到二维空间的映射，也就是说我们证明了直线上的点和平面上的点一样多（集合的势相同）。Hilbert 曲线也是一种遍历二维格点的方法，它同样可以用来证明自然数和有理数一样多。如果你已经知道此结论的 Cantor 证明，你会立刻明白 Hilbert 遍历法的证明，这里就不再多说了。当然，Hilbert 曲线也可以扩展到三维空间，甚至更高维的空间，从而建立一维到任意多维的映射关系。下图就是一个三维 Hilbert 曲线（在迭代过程中）的样子。



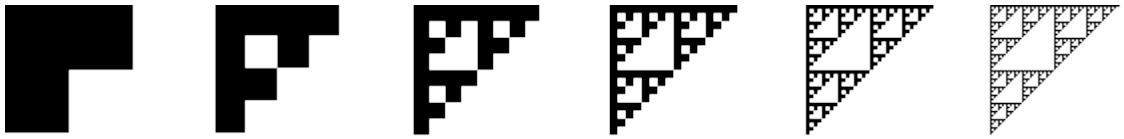
神奇的分形艺术（三）：Sierpinski 三角形

在所有的分形图形中，Sierpinski 三角形可能是大家最熟悉的了，因为它在 OI 题目中经常出现，**OJ 上的题目**和**省选题目**中都有它的身影。这篇文章将简单介绍 Sierpinski 三角形的几个惊人性质。如果你以前就对 Sierpinski 三角形有一些了解，这篇文章带给你的震撼将更大，因为你会发现 Sierpinski 三角形竟然还有这些用途。

Sierpinski 三角形的构造

和之前介绍的两种图形一样，Sierpinski 三角形也是一种分形图形，它是递归地构造的。最常见的构造方法如上图所示：把一个三角形分成四等份，挖掉中间那一份，然后继续对另外三个三角形进行这样的操作，并且无限地递归下去。每一次迭代后整个图形的面积都会减小到原来的3/4，因此最终得到的图形面积显然为0。这也就是说，Sierpinski 三角形其实是一条曲线，它的 Hausdorff 维度介于1和2之间。

Sierpinski 三角形的另一种构造方法如下图所示。把正方形分成四等份，去掉右下角的那一份，并且对另外三个正方形递归地操作下去。挖个几次后把脑袋一歪，你就可以看到一个等腰直角的 Sierpinski 三角形。



Sierpinski 三角形有一个神奇的性质：如果某一个位置上有点（没被挖去），那么它与原三角形顶点的连线上的中点处也有点。这给出另一个诡异的 Sierpinski 三角形构造方法：给出三角形的三个顶点，然后从其中一个顶点出发，每次随机向任意一个顶点移动1/2的距离（走到与那个顶点的连线的中点上），并在该位置作一个标记；无限次操作后所有的标记就组成了 Sierpinski 三角形。下面的程序演示了这一过程，程序在 fpc 2.0下通过编译。对不起用 C 语言的兄弟了，我不会 C 语言的图形操作。

```
{$ASSERTIONS+}

uses graph,crt;

const
   x1=320;  y1=20;
   x2=90;   y2=420;
   x3=550;  y3=420;
   density=2500;
   timestep=10;

var
   gd,gm,i,r:integer;
   x,y:real;

begin
   gd:=D8bit;
   gm:=m640x480;
   InitGraph(gd,gm,'');
   Assert(graphResult=grOk);

   x:=x1;
   y:=y1;
   for i:=1 to density do
   begin
      r:=random(3);
      if r=0 then
      begin
         x:=(x+x1)/2;
         y:=(y+y1)/2;
      end
      else if r=1 then
```

```
      begin
         x:=(x+x2)/2;
         y:=(y+y2)/2;
      end
      else begin
         x:=(x+x3)/2;
         y:=(y+y3)/2;
      end;
      PutPixel(round(x),round(y),white);
      Delay(timestep);
   end;
   CloseGraph;
end.
```

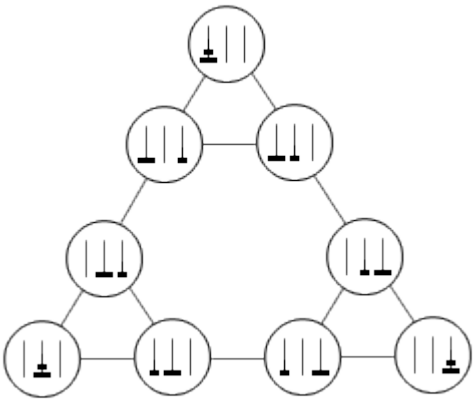
Sierpinski 三角形与杨辉三角

第一次发现 Sierpinski 三角形与杨辉三角的关系时，你会发现这玩意儿不是一般的牛。写出8行或者16行的杨辉三角，然后把杨辉三角中的奇数和偶数用不同的颜色区别开来，你会发现杨辉三角模2与 Sierpinski 三角形是等价的。也就是说，二项式系数（组合数）的奇偶性竟然可以表现为一个分形图形！在感到诧异的同时，冷静下来仔细想想，你会发现这并不难理解。

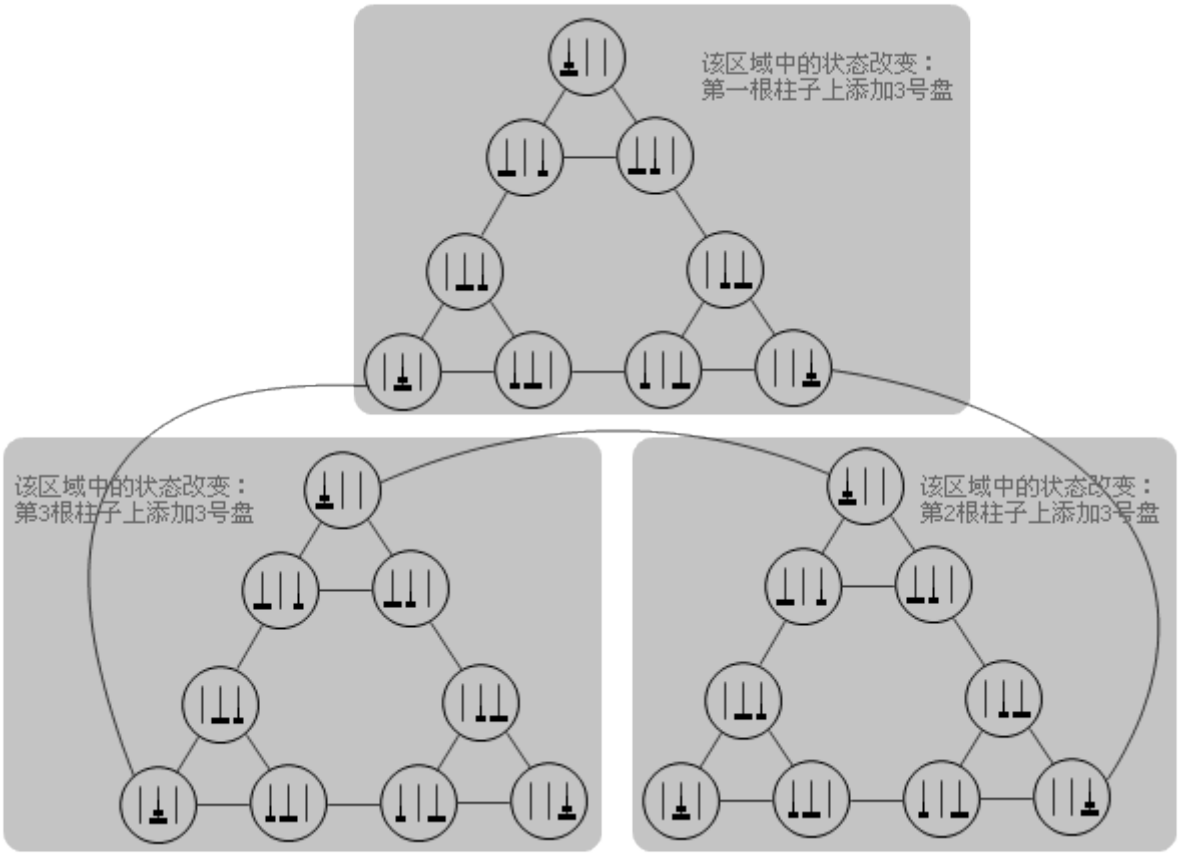
我们下面说明，如何通过杨辉三角奇偶表的前四行推出后四行来。可以看到杨辉三角的前四行是一个二阶的 Sierpinski 三角形，它的第四行全是奇数。由于奇数加奇数等于偶数，那么第五行中除了首尾两项为 1外其余项都是偶数。而偶数加偶数还是偶数，因此中间那一排连续的偶数不断地两两相加必然得到一个全是偶数项的“倒三角”。同时，第五行首尾的两个1将分别产生两个和杨辉三角前四行一样的二阶 Sierpinski 三角形。这正好组成了一个三阶的 Sierpinski 三角形。显然它的最末行仍然均为奇数，那么对于更大规模的杨辉三角，结论将继续成立。

Sierpinski 三角形与 Hanoi 塔

有没有想过，把 Hanoi 塔的所有状态画出来，可以转移的状态间连一条线，最后得到的是一个什么样的图形？二阶 Hanoi 塔反正也只有9个节点，你可以自己试着画一下。不断调整节点的位置后，得到的图形大概就像这个样子：



如果把三阶的 Hanoi 塔表示成无向图的话，得到的结果就是三阶的 Sierpinski 三角形。下面的这张图说明了这一点。把二阶 Hanoi 塔对应的无向图复制两份放在下面，然后在不同的柱子上为每个子图的每个状态添加一个更大的盘子。新的图中原来可以互相转移的状态现在仍然可以转移，同时还出现了三个新的转移关系将三个子图连接在了一起。重新调整一下各个节点的位置，我们可以得到一个三阶的 Sierpinski 三角形。



显然，对于更大规模的 Hanoi 塔问题，结论仍然成立。

Sierpinski 三角形与位运算

编程画出 Sierpinski 三角形比想象中的更简单。下面的两个代码（实质相同，仅语言不同）可以打印出一个 Sierpinski 三角形来。

```
const
  n=1 shl 5-1;
var
  i,j:integer;
begin
  for i:=0 to n do
  begin
    for j:=0 to n do
      if i and j = j then write('#')
      else write(' ');
    writeln;
  end;
  readln;
end.
```

```
#include <stdio.h>
int main()
{
   const int n=(1<<5)-1;
   int i,j;
   for (i=0; i<=n; i++)
   {
      for (j=0; j<=n; j++)
        printf( (i&j)==j ? "#" : " ");
      printf("\n");
```

```
    }
    getchar();
    return 0;
}
```

上面两个程序是一样的。程序将输出：

```
#
##
# #
####
#   #
##  ##
# # # #
########
#       #
##      ##
# #     # #
####    ####
#   #   #   #
##  ##  ##  ##
# # # # # # # #
################
#               #
##              ##
# #             # #
####            ####
#   #           #   #
##  ##          ##  ##
# # # #         # # # #
########        ########
#       #       #       #
##      ##      ##      ##
# #     # #     # #     # #
####    ####    ####    ####
#   #   #   #   #   #   #   #
##  ##  ##  ##  ##  ##  ##  ##
# # # # # # # # # # # # # # # #
################################
```

这个程序告诉我们：在第 i 行第 j 列上打一个点当且仅当 i and j=j，这样最后得到的图形就是一个 Sierpinski 三角形。这是为什么呢？其实原因很简单。把 i 和 j 写成二进制（添加前导0使它们位数相同），由于 j 不能大于 i，因此只有下面三种情况：

情况一：

i = 1?????

j = 1?????

问号部分 i 大于等于 j

i 的问号部分记作 i'，j 的问号部分记作 j'。此时 i and j=j 当且仅当 i' and j'=j'

情况二：

i = 1?????

j = 0?????

问号部分 i 大于等于 j

i 的问号部分记作 i'，j 的问号部分记作 j'。此时 i and j=j 当且仅当 i' and j'=j'

情况三：

i = 1?????

j = 0?????

问号部分 i 小于 j

此时 i and j 永远不可能等于 j。i' < j'意味着 i'和 j'中首次出现数字不同的那一位上前者为0，后者为1，那么 i 和 j 做 and 运算时这一位的结果是0，与 j 不等。

注意到，去掉一个二进制数最高位上的“1”，相当于从这个数中减去不超过它的最大的2的幂。观察每一种情况中 i,j 和 i',j'的实际位置，不难发现这三种情况递归地定义出了整个 Sierpinski 三角形。

嘿！发现没有，我通过 Sierpinski 三角形证明了这个结论：组合数 C(N,K)为奇数当且仅当 N and K=K。这篇文章很早之前就计划在写了，前几天有人问到这个东西，今天顺便也写进来。

另外，把 i and j=j 换成 i or j=n 也可以打印出 Sierpinski 三角形来。i and j=j 表示 j 的二进制中有1的位置上 i 也有个1，那么此时 i or (not j)结果一定全为1（相当于程序中的常量 n），因此打印出来的结果与原来的输出正好左右镜像。

神奇的分形艺术（四）：Julia 集和 Mandelbrot 集

考虑函数 f(z)=z^2-0.75。固定 z0的值后，我们可以通过不断地迭代算出一系列的 z 值：z1=f(z0), z2=f(z1), z3=f(z2), ...。比如，当 z0 = 1时，我们可以依次迭代出：

z1 = f(1.0) = 1.0^2 - 0.75 = 0.25

z2 = f(0.25) = 0.25^2 - 0.75 = -0.6875

z3 = f(-0.6875) = (-0.6875)^2 - 0.75 = -0.2773

z4 = f(-0.2773) = (-0.2773)^2 - 0.75 = -0.6731

z5 = f(-0.6731) = (-0.6731)^2 - 0.75 = -0.2970

...

可以看出，z 值始终在某一范围内，并将最终收敛到某一个值上。

但当 z0=2时，情况就不一样了。几次迭代后我们将立即发现 z 值最终会趋于无穷大：

z1 = f(2.0) = (2.0)^2 - 0.75 = 3.25

z2 = f(3.25) = (3.25)^2 - 0.75 = 9.8125

z3 = f(9.8125) = (9.8125)^2 - 0.75 = 95.535

z4 = f(95.535) = (95.535)^2 - 0.75 = 9126.2

z5 = f(9126.2) = (9126.2)^2 - 0.75 = 83287819.2

...

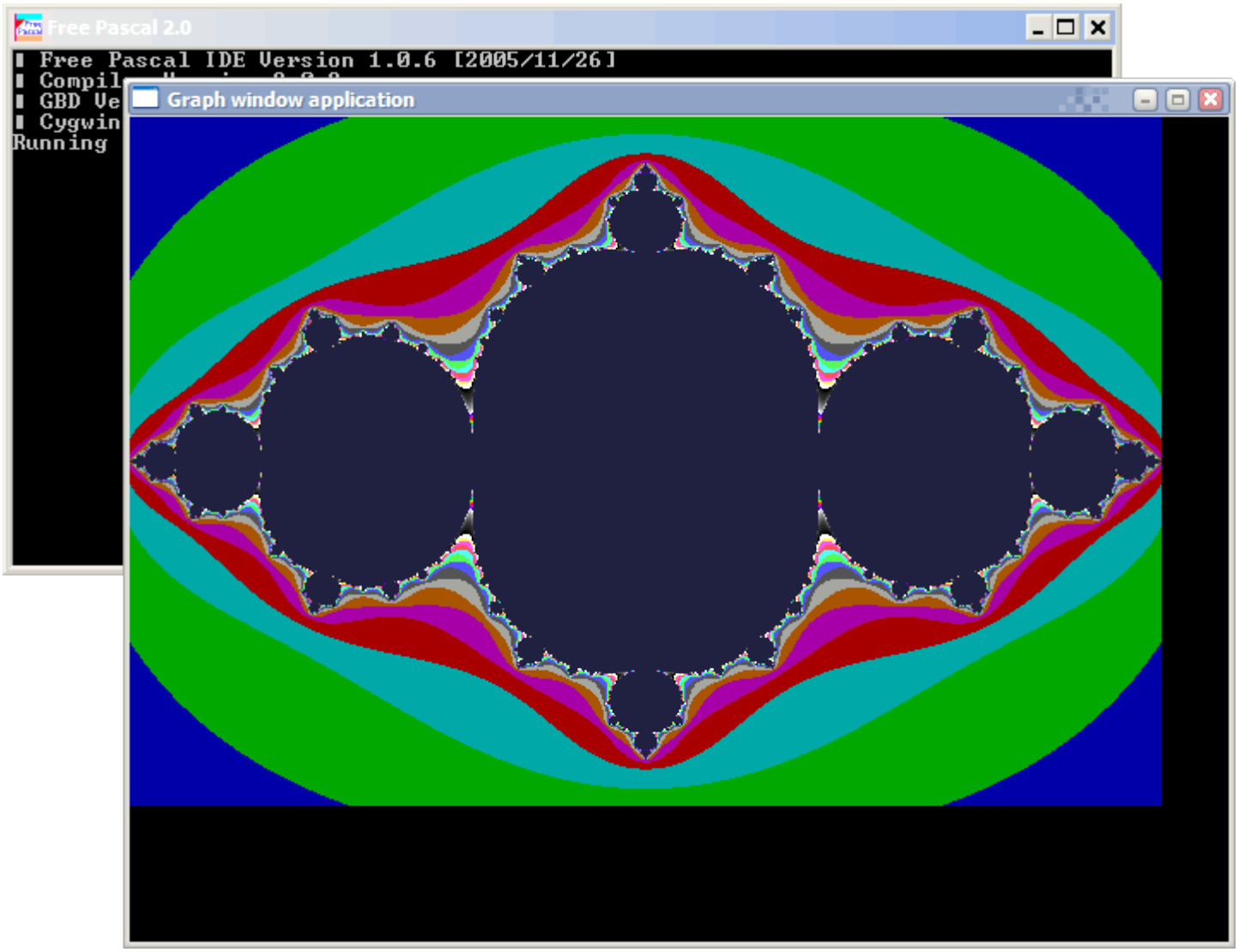
经过计算，我们可以得到如下结论：当 z0属于[-1.5, 1.5]时，z 值始终不会超出某个范围；而当 z0小于-1.5或大于1.5后，z 值最终将趋于无穷。

现在，我们把这个函数扩展到整个复数范围。对于复数 z0=x+iy，取不同的 x 值和 y 值，函数迭代的结果不一样：对于有些 z0，函数值约束在某一范围内；而对于另一些 z0，函数值则发散到无穷。由于复数对应平面上的点，因此我们可以用一个平面图形来表示，对于哪些 z0函数值最终趋于无穷，对于哪些 z0函数值最终不会趋于无穷。我们用深灰色表示不会使函数值趋于无穷的 z0；对于其它的 z0，我们用不同的颜色来区别不同的发散速度。由于当某个时候|z|>2时，函数值一定发散，因此这里定义发散速度为：使|z|大于2的迭代次数越少，则发散速度越快。这个图形可以编程画出。和上次一样，我用 Pascal 语言，因为我不会 C 的图形操作。某个 MM 要过生日了，我把这个自己编程画的图片送给她^_^

```
{$ASSERTIONS+}

uses graph;
```

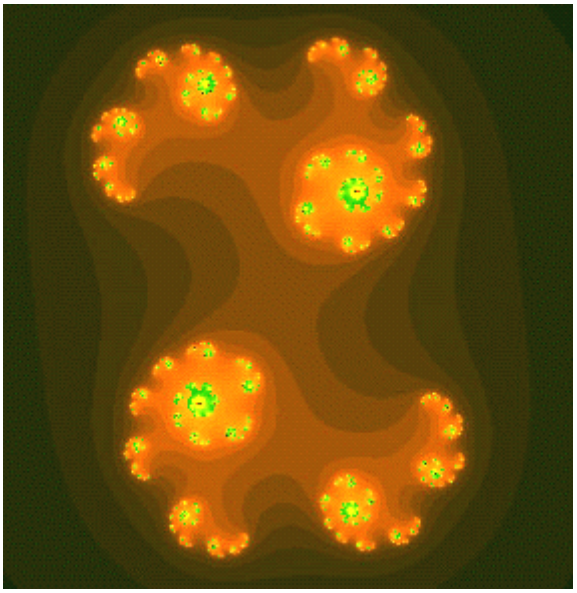
```
type
  complex=record
    re:real;
    im:real;
  end;

operator * (a:complex; b:complex) c:complex;
begin
  c.re := a.re*b.re - a.im*b.im;
  c.im := a.im*b.re + a.re*b.im;
end;

operator + (a:complex; b:complex) c:complex;
begin
  c.re := a.re + b.re;
  c.im := a.im + b.im;
end;

var
  z,c:complex;
  gd,gm,i,j,k:integer;
begin
  gd:=D8bit;
  gm:=m640x480;
  InitGraph(gd,gm,'');
  Assert(graphResult=grOk);

  c.re:=-0.75;
  c.im:=0;
  for i:=-300 to 300 do
  for j:=-200 to 200 do
  begin
    z.re:=i/200;
    z.im:=j/200;
    for k:=0 to 200 do
    begin
      if sqrt(z.re*z.re + z.im*z.im) >2 then break
      else z:=(z*z)+c;
    end;
    PutPixel(i+300,j+200,k)
  end;

  readln;
  CloseGraph;
end.
```

代码在 Windows XP SP2，FPC 2.0下通过编译，麻烦大家帮忙报告一下程序运行是否正常（上次有人告诉我说我写的绘图程序不能编译）。在我这里，程序运行的结果如下：

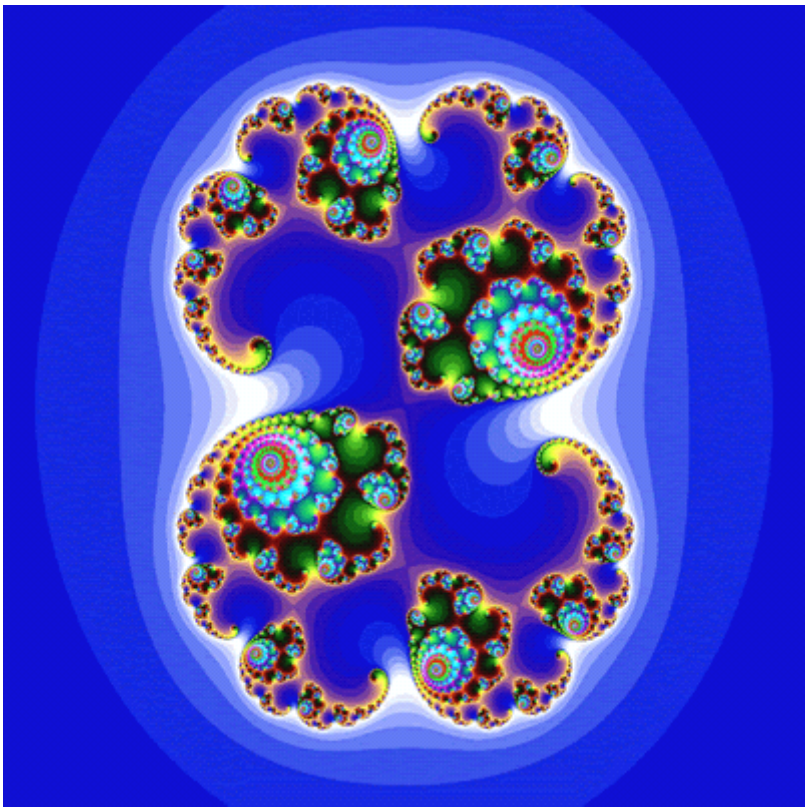


这个美丽的分形图形表现的就是 f(z)=z^2-0.75时的 Julia 集。考虑复数函数 f(z)=z^2+c，不同的复数 c 对应着不同的 Julia 集。也就是说，每取一个不同的 c 你都能得到一个不同的 Julia 集分形图形，并且令人吃惊的是每一个分形图形都是那么美丽。下面的六幅图片是取不同的 c 值得到的分形图形。你可能不相信这样一个简单的构造法则可以生成这么美丽的图形，这没什么，你可以改变上面程序代码中 c 变量的值来亲自验证。

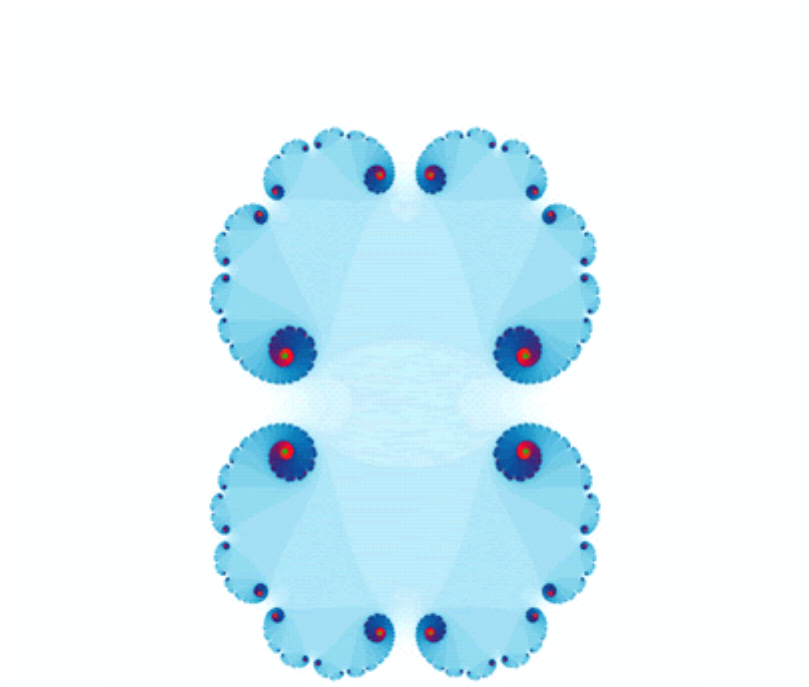
c = 0.45, -0.1428



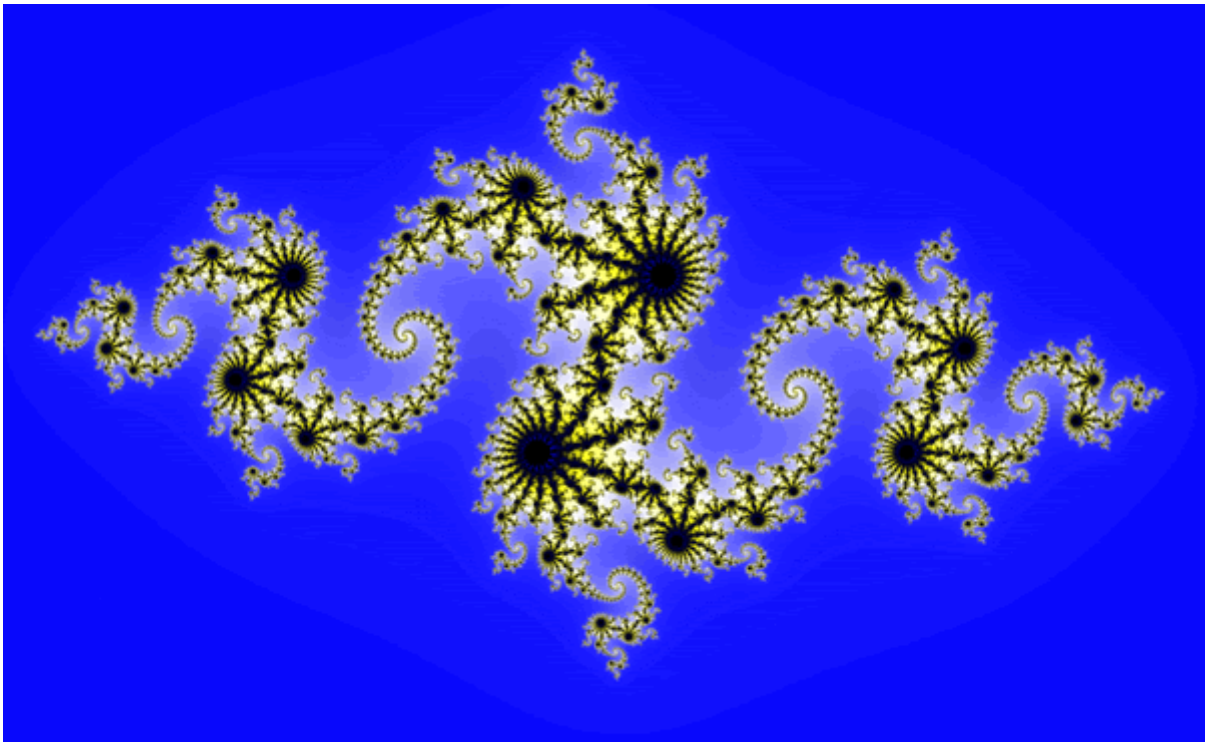
c = 0.285, 0.01



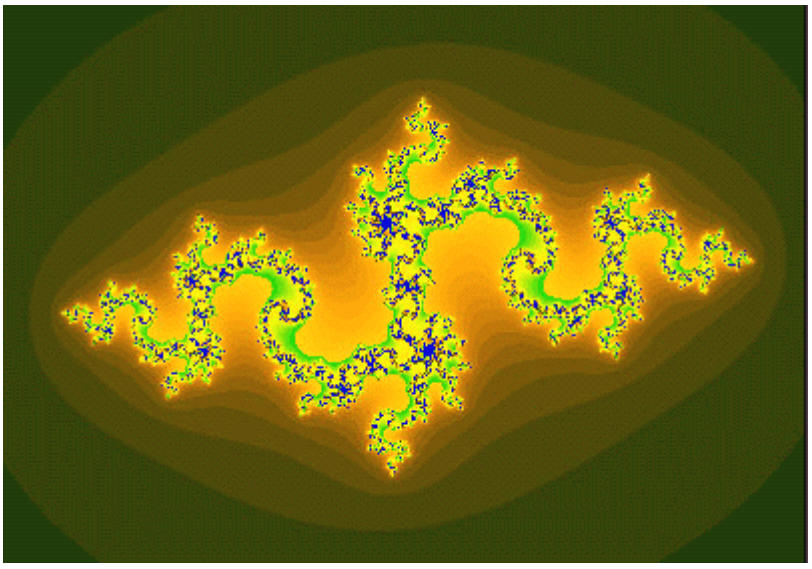
c = 0.285, 0



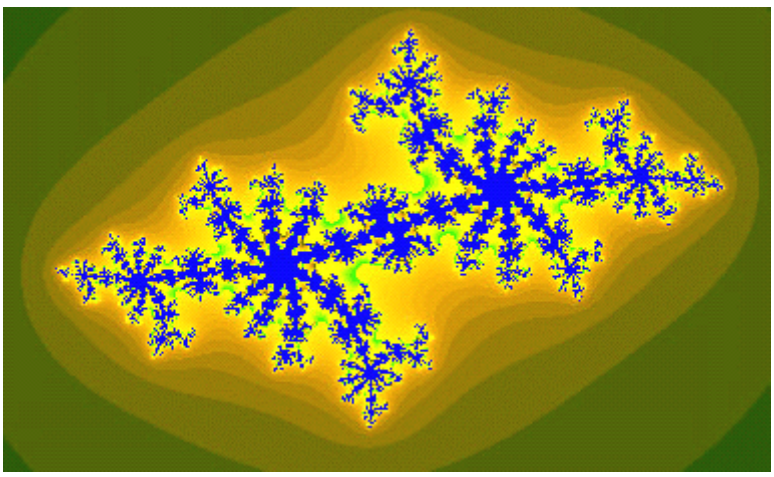
c = -0.8, 0.156



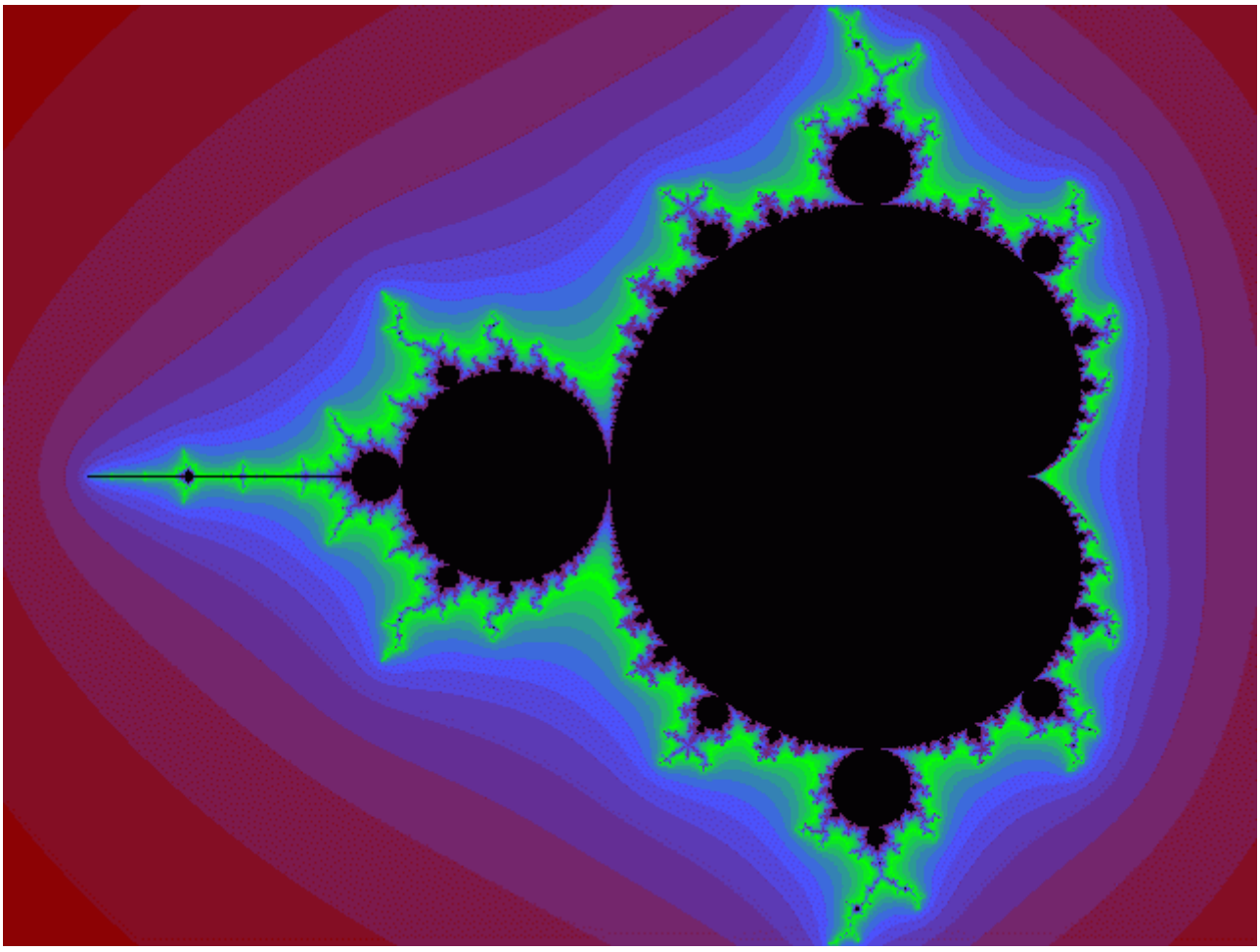
c = -0.835, -0.2321



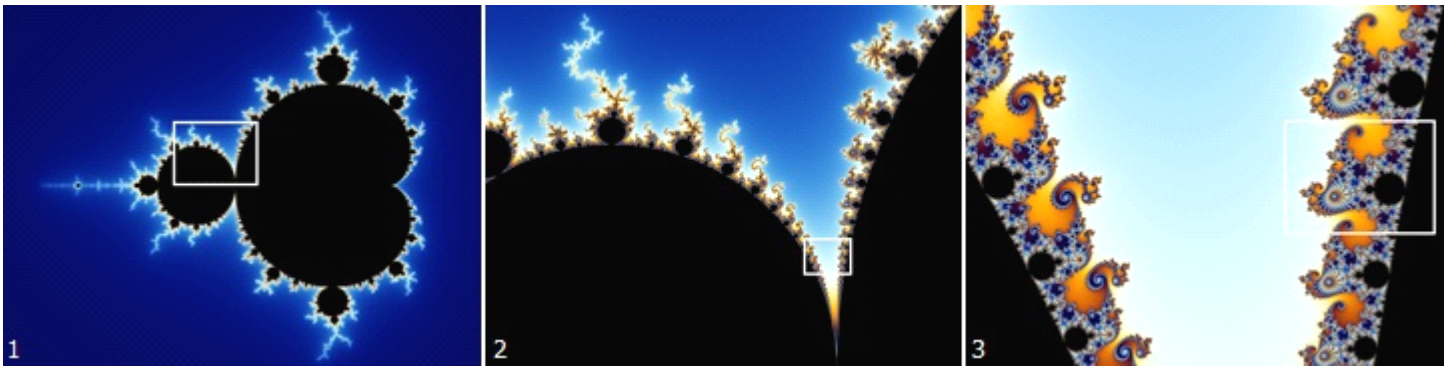
c = -0.70176, -0.3842

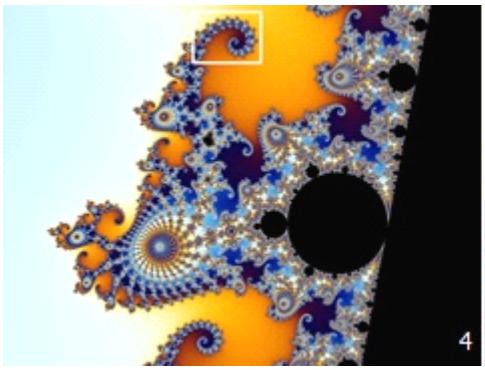


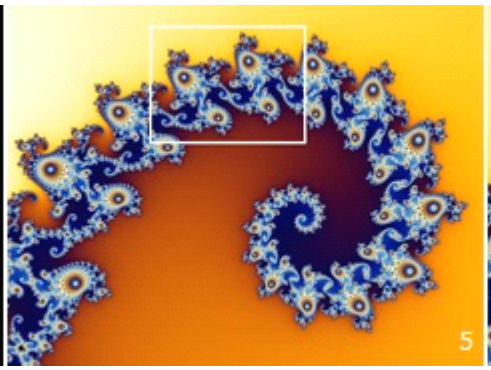
类似地，我们固定 z0=0，那么对于不同的复数 c，函数的迭代结果也不同。由于复数 c 对应平面上的点，因此我们可以用一个平面图形来表示，对于某个复数 c，函数 f(z)=z^2+c 从 z0=0开始迭代是否会发散到无穷。我们同样用不同颜色来表示不同的发散速度，最后得出的就是 Mandelbrot 集分形图形：

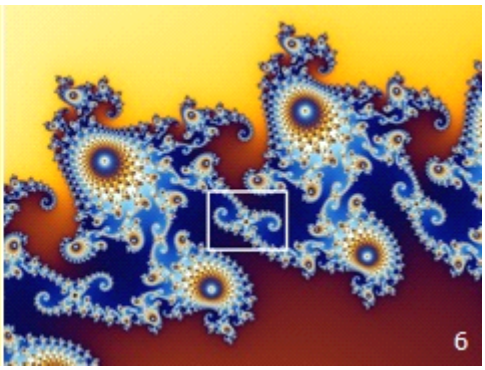


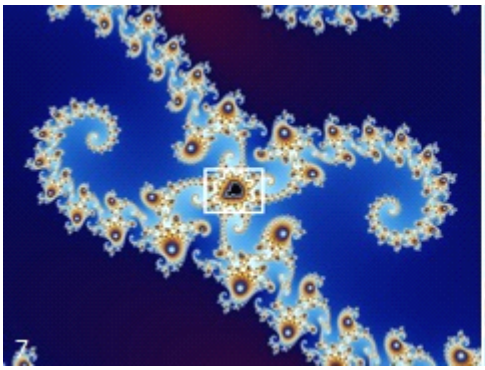
前面说过，分形图形是可以无限递归下去的，它的复杂度不随尺度减小而消失。Mandelbrot 集的神奇之处就在于，你可以对这个分形图形不断放大，不同的尺度下你所看到的景象可能完全不同。放大到一定时候，你可以看到更小规模的 Mandelbrot 集，这证明 Mandelbrot 集是自相似的。下面的15幅图演示了 Mandelbrot 集的一个放大过程，你可以在这个过程中看到不同样式的分形图形。

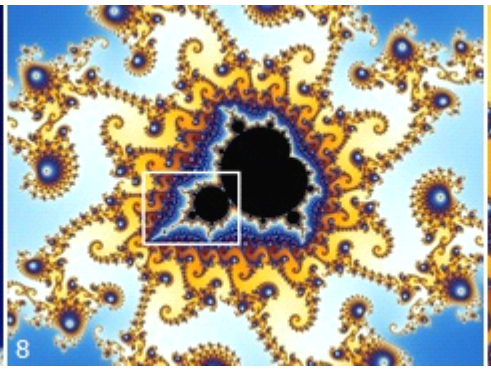


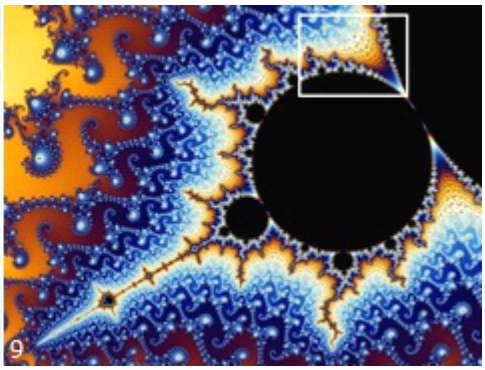

4

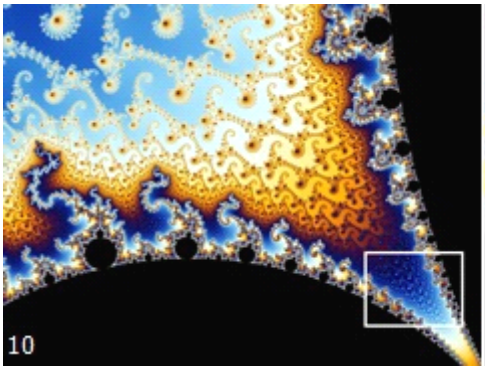

5

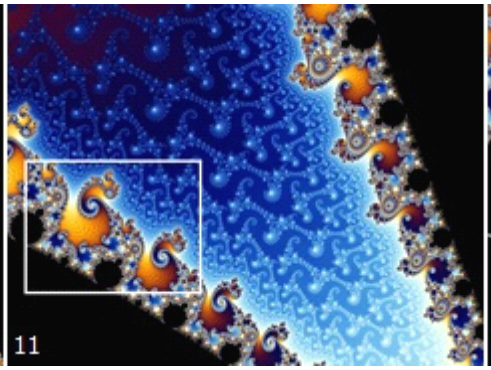

6

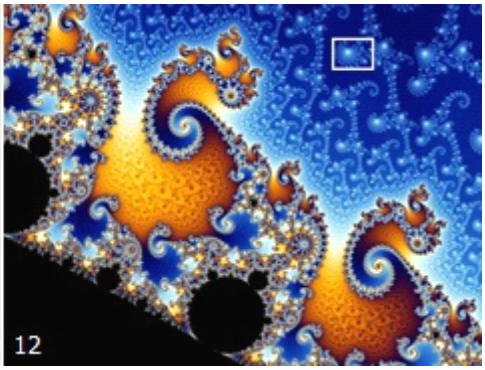

7

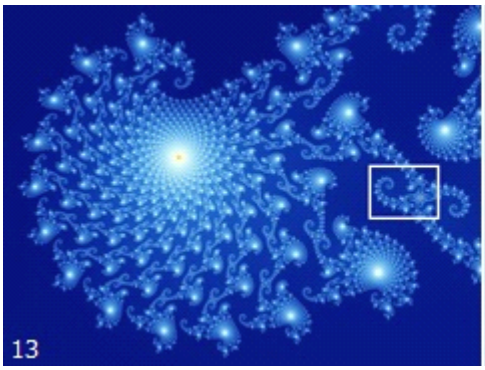

8

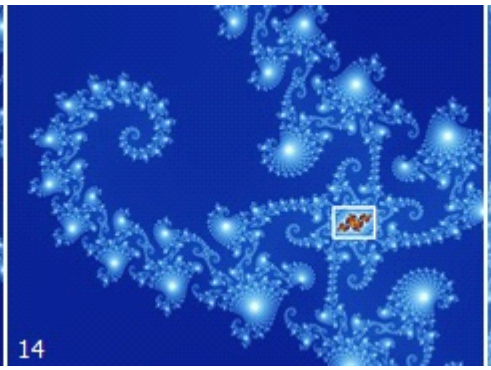

9

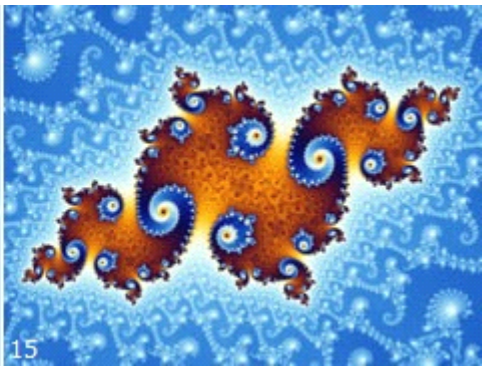

10


11


12


13


14


15