Leobardo Rosales

Final

math 107B

This project is concerned with several examples of sets and functions which can be constructed iteratively. These sets and functions are important ones, for they have been the source of examples and counter-examples for analysts and topologists. As such, this project deals not only with the technical challenges of reproducing the construction of these examples, but we will also discuss their relevance.

We start with an interesting example, the Cantor Set and function.

Cantor Set

The Cantor Set is a subset of the interval [0,1] which was discovered by Georg Cantor and presented in the appendix of a paper written in 1883. The set is constructed as follows, we start with the interval [0,1]. We remove the open middle segment (1/3,2/3) from this interval, and we are left with two closed segements, [0,1/3] and [2/3,1]. We grab each of these segments and remove the middle thirds from them, so that we are left with the union of the closed intervals [0,1/9],[2/9,1/3],[2/3,7/9], and [8/9,1]. We then continue this process. What is left is what we call the Cantor Set.

This set has remarkable properties. Firstly, we should remark that this set is non-empty. We always remove the middle thirds of intervals in constructing the set, and as such, we will never remove the endpoints of any segment from the previous step of the construction. Thus, the points 0,1/9,2/9,1/3 and so on will be in the Cantor Set. However, much more is left than just these endpoints, in fact so much is left over as to make the Cantor Set uncountable. The proof is simple, once we identify that the Cantor Set is the set of all number in [0,1] whose ternary decimal expansion contains only 0's and 1's. Such a set is uncountable by a diagonalization argument. For supposing that we had such an enumeration of numbers, then we could list them as a sequence of points a**(n)** in their ternary represenation as follows:

a(0)= .020200020020...
a(1)= .220220020000...
a(2)= .000022222022...

and so forth. However, then by letting **A** be the number whose **n**th decimal place is a 0 if the **n**th decimal place of **a(n)** is 2, and vice versa, we get that **A** is in the Cantor Set but is different from any **a(n)** in our enumeration. Thus, the Cantor Set must be uncountable.

The Cantor Set is also compact, perfect, and its closure has an empty interior. That the Cantor Set is compact is easy to see, for it is the countable intersection of the compact sets [0,1/3], [2/3,1], [0,1/9], [2/9,1/3], [2/3,7/9], [8/9,1], and so forth. The Cantor Set is also totally disconnected, meaning that we cannot find two distinct points in the Cantor

Set that can be connected by a continuous function on [0,1] whose image lies totally inside the Cantor Set. Thus, the Cantor Set is in some respects very meaty. However, in other respects the Cantor Set is also very thin, in particular with respect to measure which we shall discuss later.

Now, how can we use maple to better understand the Cantor Set? My first goal was to devise an algorithm which given and **i** and a **k**, gives the **i**th point which is an endpoint of a segment removed at the **k**th step of the construction of the Cantor Set. The algorithm is as follows:

```
> cantorpoint:= proc(i,k)
option remember ;
if i=1 then 0 else
if i mod 2 = 0
then cantorpoint(i-1,k)+(1/(3^k)) ;
else cantorpoint(i-1,k)+((3^((ifactors(i-1)[2][1][2])-
1))/(3^k)) ;
fi fi end;
```

```
cantorpoint := proc(i, k)
option remember;
    if i = 1 then 0
    else
        if i mod 2 = 0 then cantorpoint(i - 1, k) + 1/3^k
        else cantorpoint(i - 1, k) + 3^(ifactors(i - 1)[2][1][2] - 1)/3^k
        end if
    end if
end proc
```

The algorithm is recursive. The **i**th point is given by adding an appropriate amount to the previous point, what you add depends highly on the divisibility of **i** by 2. In this algorithm, for each **k**, **i** ranges from 1 to 2^(**k**+1).

To verify the validity of this algorithm, let us make a list of the cantor points:

```
> cantorsequence:=(k)->
[seq(cantorpoint(i,k),i=1..2^(k+1))];
```

```
cantorsequence := k → [seq(cantorpoint(i, k), i = 1 .. 2^(k + 1))]
```

and let us thus list the points used in the construction of the cantor set for the steps 1,2,3:

```
> cantorsequence(1);
```

$$\left[0, \frac{1}{3}, \frac{2}{3}, 1\right]$$

```
> cantorsequence(2);
```

$$\left[0, \frac{1}{9}, \frac{2}{9}, \frac{1}{3}, \frac{2}{3}, \frac{7}{9}, \frac{8}{9}, 1\right]$$

```
> cantorsequence(3);
```

$$\left[0, \frac{1}{27}, \frac{2}{27}, \frac{1}{9}, \frac{2}{9}, \frac{7}{27}, \frac{8}{27}, \frac{1}{3}, \frac{2}{3}, \frac{19}{27}, \frac{20}{27}, \frac{7}{9}, \frac{8}{9}, \frac{25}{27}, \frac{26}{27}, 1\right]$$

This all checks out. We see that on the second step of the construction, for example, after having removed the segment (1/3,2/3), we then remove (1/9,2/9) and (7/9,8/9). Thus **cantorsequence**(2) gives all of the important points of our construction, including 0 and 1. So, now let us make a procedure that gives us what is left at the **k**th step of the Cantor Set construction. Specifically, given **k**=1 this procedure will tell us that we have removed the segment (1/3,2/3) from [0,1], leaving us with the union of the closed segments [0,1/3] and [2/3,1]. We will do so by making a procedure that spits out a list of lists which is to represent a union of closed intervals. Thus in the case for **k**=1, this procedure will give [[0,1/3], [2/3,1]] as the answer. So:

```
> cantorset:=(k)->[seq([cantorpoint(2*i-
1,k),cantorpoint(2*i,k)],i=1..(2^k))];
```

$$cantorset := k \rightarrow [\mathrm{seq}([\mathrm{cantorpoint}(2\,i-1, k), \mathrm{cantorpoint}(2\,i, k)], i = 1 .. 2^k)]$$

and now let us see what is left after steps 1,2,3,and 4:

```
> cantorset(1);
```

$$\left[\left[0, \frac{1}{3}\right], \left[\frac{2}{3}, 1\right]\right]$$

```
> cantorset(2);
```

$$\left[\left[0, \frac{1}{9}\right], \left[\frac{2}{9}, \frac{1}{3}\right], \left[\frac{2}{3}, \frac{7}{9}\right], \left[\frac{8}{9}, 1\right]\right]$$

```
> cantorset(3);
```

$$\left[\left[0, \frac{1}{27}\right], \left[\frac{2}{27}, \frac{1}{9}\right], \left[\frac{2}{9}, \frac{7}{27}\right], \left[\frac{8}{27}, \frac{1}{3}\right], \left[\frac{2}{3}, \frac{19}{27}\right], \left[\frac{20}{27}, \frac{7}{9}\right], \left[\frac{8}{9}, \frac{25}{27}\right], \left[\frac{26}{27}, 1\right]\right]$$

```
> cantorset(4);
```

$$\left[\left[0, \frac{1}{81}\right], \left[\frac{2}{81}, \frac{1}{27}\right], \left[\frac{2}{27}, \frac{7}{81}\right], \left[\frac{8}{81}, \frac{1}{9}\right], \left[\frac{2}{9}, \frac{19}{81}\right], \left[\frac{20}{81}, \frac{7}{27}\right], \left[\frac{8}{27}, \frac{25}{81}\right], \left[\frac{26}{81}, \frac{1}{3}\right], \left[\frac{2}{3}, \frac{55}{81}\right], \left[\frac{56}{81}, \frac{19}{27}\right], \left[\frac{20}{27}, \frac{61}{81}\right], \left[\frac{62}{81}, \frac{7}{9}\right], \left[\frac{8}{9}, \frac{73}{81}\right], \left[\frac{74}{81}, \frac{25}{27}\right], \left[\frac{26}{27}, \frac{79}{81}\right], \left[\frac{80}{81}, 1\right]\right]$$

Now we may illustrate another remarkable fact about the Cantor Set, which is that it has Lebesgue measure zero. We can see this most readibly by investigating the measure of the complement of the Cantor Set on [0,1]. In the first step of the construction, we remove the open interval (1/3,2/3), which has measure 1/3. At the second step, we remove two disjoint intervals of length 1/9, each of which is disjoint from the first interval removed. We proceed, at each **k**th step removing 2^(**k**-1) disjoint intervals that are also disjoint from anything else removed, that are of length 3^**k**. Since all of the segments are mutually disjoint, we may get the measure of the complement of the Cantor Set by adding all of the measures of each of the removed segments, which in light of my previous statements, can be evaluated as the infinite sum starting from 1 of 2^(**k**-1)/3^**k**. This is a geometric series which evaluates to 1, the measure of [0,1], so that the Cantor Set must have zero measure.

Let us test this through data. First, we write a program which given a disjoint union of sets represented by a list of two element lists, will give the measure of that set by merely subtracting endpoints and totalling the results:

```
> measure:= proc(x)
L:=0 ;
for i from 1 to nops(x) do L:=L+(x)[i][2]-(x)[i][1] od ;
L ;
end;
```

```
Warning, `L` is implicitly declared local to procedure `measure`

Warning, `i` is implicitly declared local to procedure `measure`
```

```
measure := proc(x)
local L, i;
    L := 0; for i to nops(x) do L := L + x[i][2] - x[i][1] end do ; L
end proc
```

Then by calculating some examples, we can see the behavior of the measure of each step of the construction, which should go to zero:

```
> measure(cantorset(1));
```

$$\frac{2}{3}$$

```
> measure(cantorset(2));
```

$$\frac{4}{9}$$

```
> measure(cantorset(3));
```

$$\frac{8}{27}$$

```
> evalf(measure(cantorset(4)));
                              .1975308642

> evalf(measure(cantorset(8)));
                              .03901844231

> evalf(measure(cantorset(15)));
                              .002283658261
```

Now we would like a way to visualize the steps taken in constructing the Cantor Set. We would like to construct a function such that given a **k**, this function is 1 on the segments removed up to the **k**th step of the construction of the Cantor Set, and is zero everywhere else.

Conceptually this is no problem at all, given that we have already devised a program which gives the cantor points. The idea is that at each step **k**, what is removed lies between **cantorpoint(i,k)** and **cantorpoint(i+1,k)** for each **i** between 1 and 2^(**k**+1)-1

that is even. Thus, ideally we should define our function, which we shall call **ComplementFunction(x,k)** as

ComplementFunction(x,k)= 1 if for some even **i** between 1 and 2^(**k**+1)-1 , **cantorpoint(i,k)** < x < **cantorpoint(i+1,k)**
0 everywhere else.

Now is where we run into some difficulties. Basically, we are attempting to define a sequence of functions that are defined piecewise. However, the number of pieces varies with **k**. Even though maple does provide one with immediate tools in order to define piecewise functions, it does not allow for the pieces to vary. In other words, we can only define functions where the pieces are set and definite.

This nothwithstanding, there are two methods we can use to get around this obstruction. One is involving sums, which is what we will use now, and the other involves do-loops. What we can do is define a function of x,**i**, and **k** which is 1 if **cantorpoint(**2***i,k)** < x < **cantorpoint(**2***i+1,k)**. In essence, for a fixed **k**, this function draws each of the pieces where our ComplementFunction will be 1, individually. Everywhere else this complementfunction will be zero.

```
> complementfunction:= proc(x,i,k)
if cantorpoint(2*i,k) < x and x < cantorpoint(2*i+1,k)
then 1 else 0
fi end ;
```

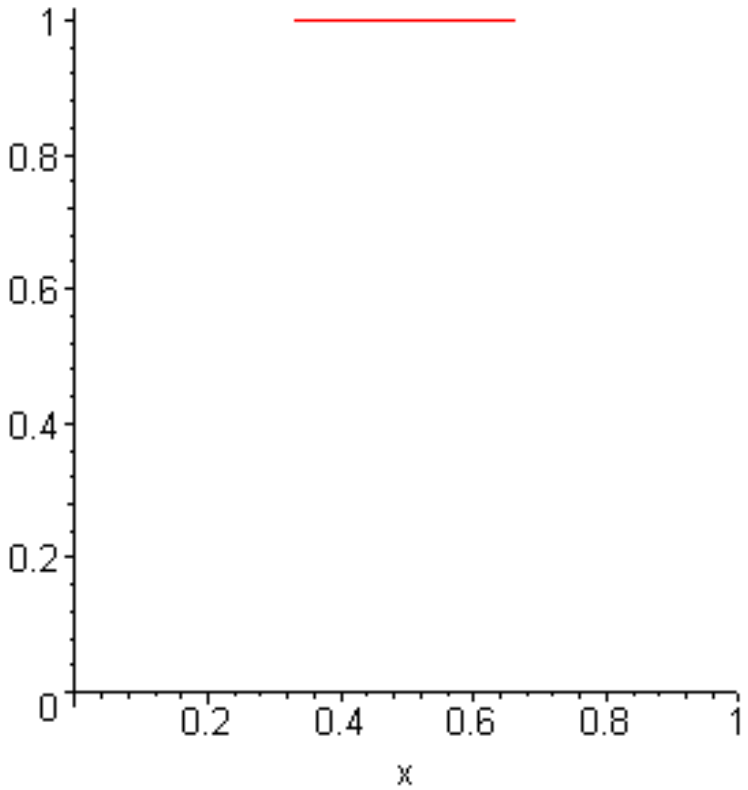
complementfunction := **proc** (*x*, *i*, *k*)

if cantorpoint(2×*i*, *k*) < *x* **and** *x* < cantorpoint(2×*i* + 1, *k*) **then** 1 **else** 0 **end if**

end proc

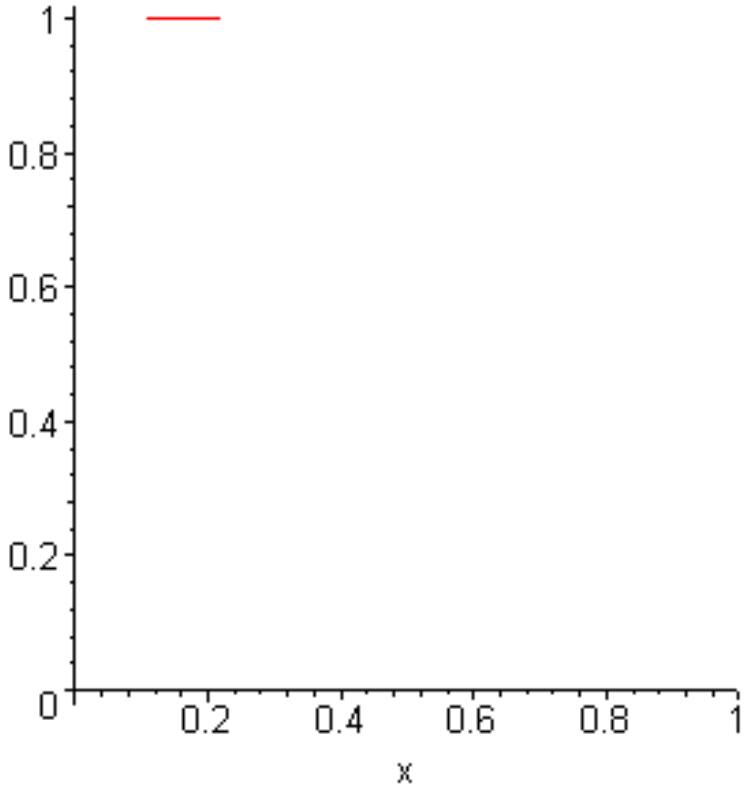
So, for example, for **k**=1 we only removed one segment, and so there will be only one pice to graph at **i**=1:

```
> plot('complementfunction(x,1,1)',x=0..1,discont = true );
```

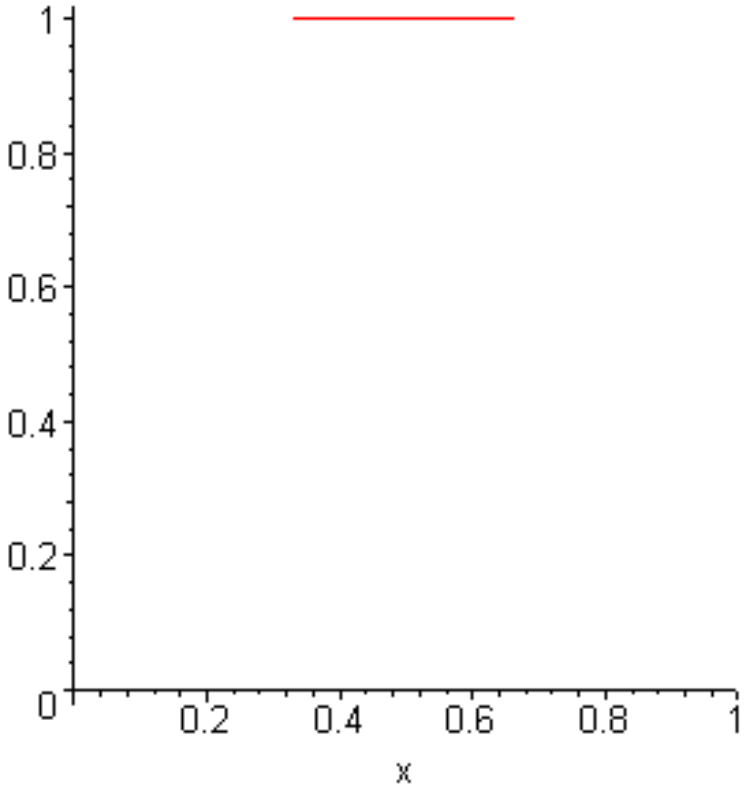


Up to step **k**=2, we removed three segments. So complementfunction should draw three heights at **i**=1,2 and 3:

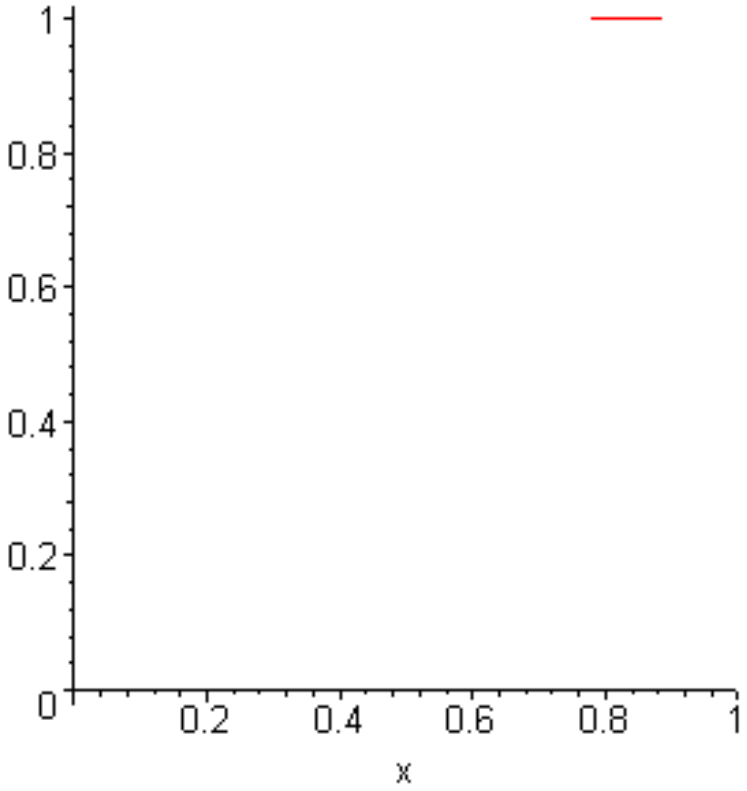
> **`plot('complementfunction(x,1,2)',x=0..1,discont = true );`**



```
> plot('complementfunction(x,2,2)',x=0..1,discont = true );
```




```
> plot('complementfunction(x,3,2)',x=0..1,discont = true );
```



Notice that the graph at **i**=2,**k**=2 is the same for **i**=1,**k**=1.

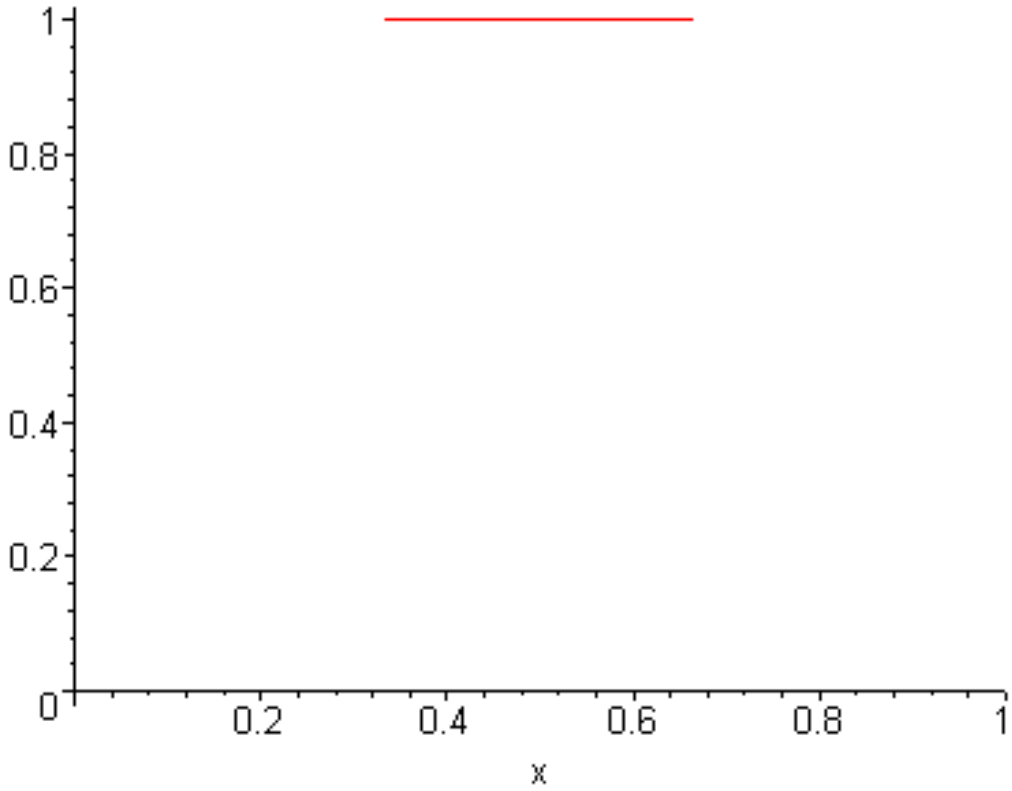
Now, to get **ComplementFunction**, we do the clever thing and sum **complementfunction** for all the values of **i** that give us all of the even numbers between 1 and 2^(**k**+1)-1. This gives:

```
> ComplementFunction:= (x, k) ->
sum('complementfunction(x,i,k)', 'i'=1..((2^k)-1));
```

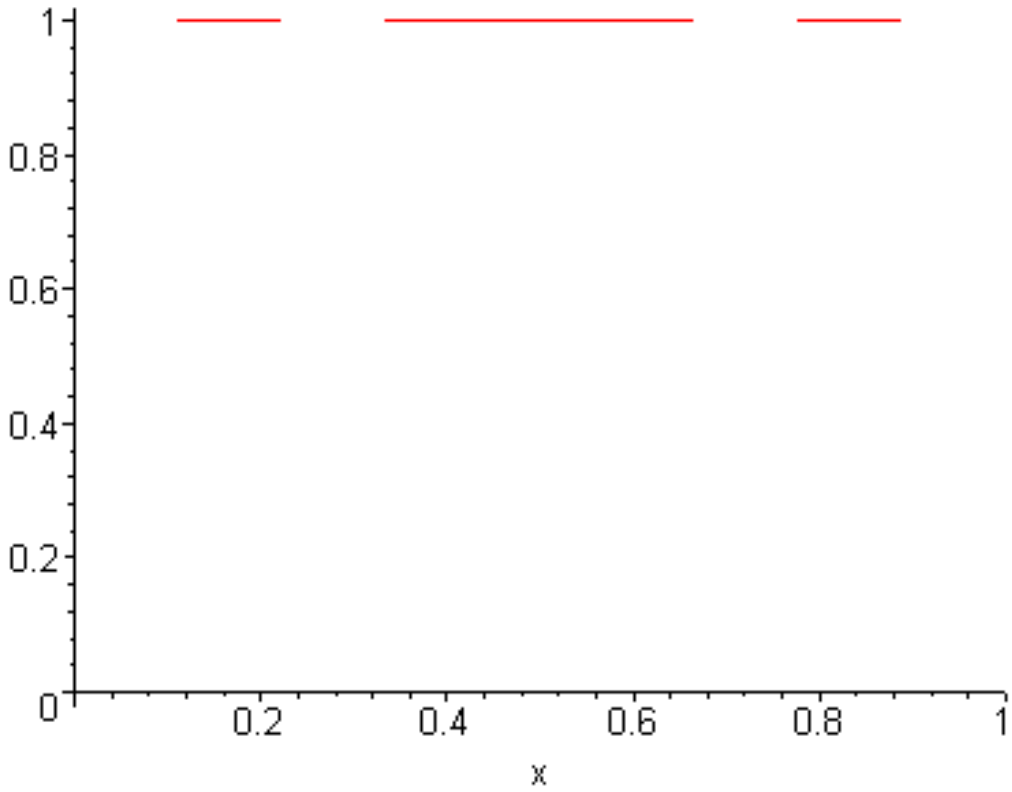
$$ComplementFunction := (x, k) \rightarrow \sum_{'i'=1}^{2^k-1} \text{'complementfunction}(x, i, k)\text{'}$$

To make an analogy, what we have done is build the four walls of a house seperately, and then join them together in the end. Let us plot **ComplementFunction** for several **k**:

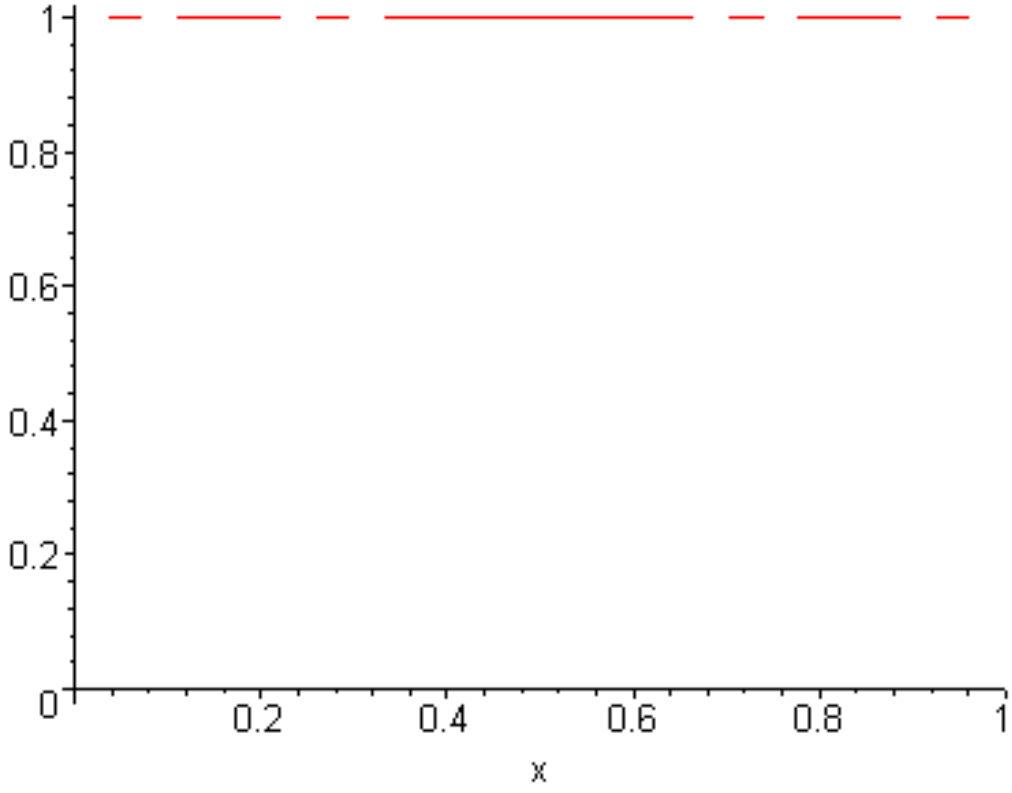
```
> plot('ComplementFunction(x,1)',x=0..1,discont = true );
```



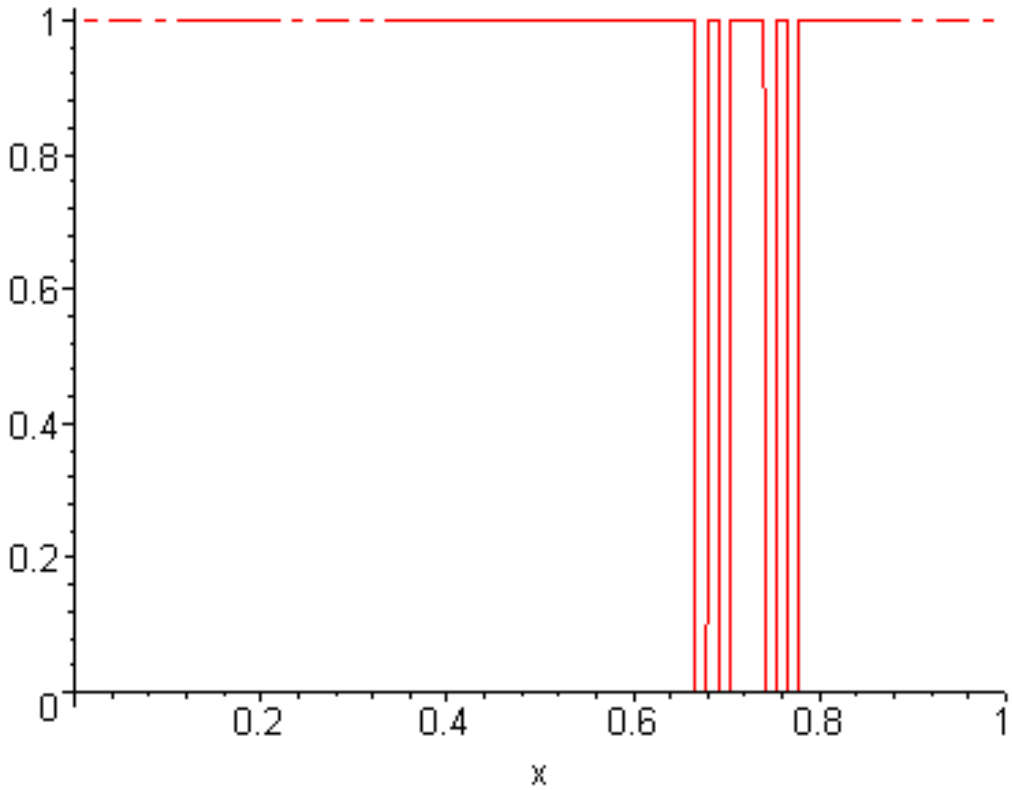

```
> plot('ComplementFunction(x,2)',x=0..1,discont = true );
```




```
> plot('ComplementFunction(x,3)',x=0..1,discont = true );
```

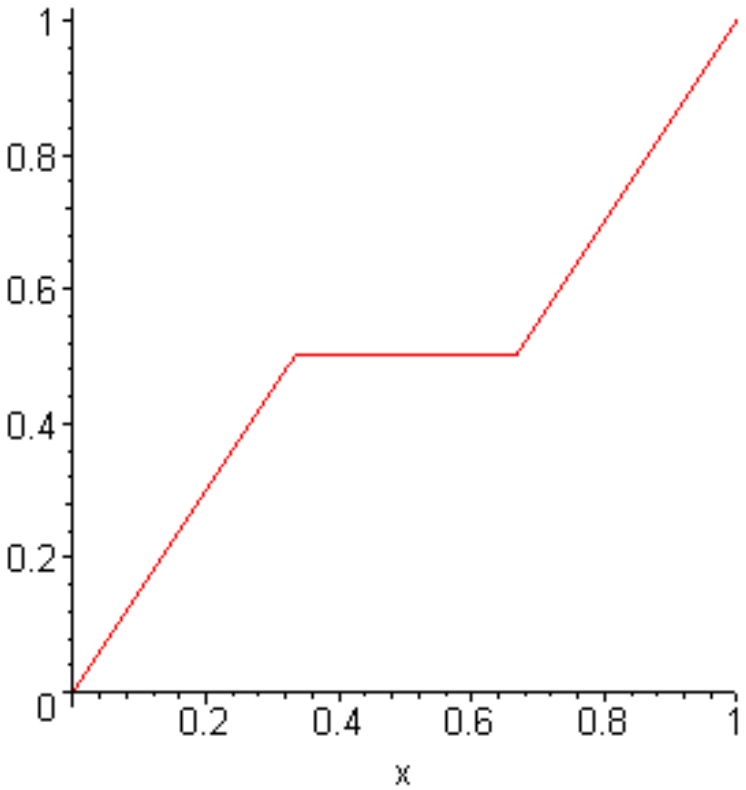



```
> plot('ComplementFunction(x,4)',x=0..1,discont = true);
```

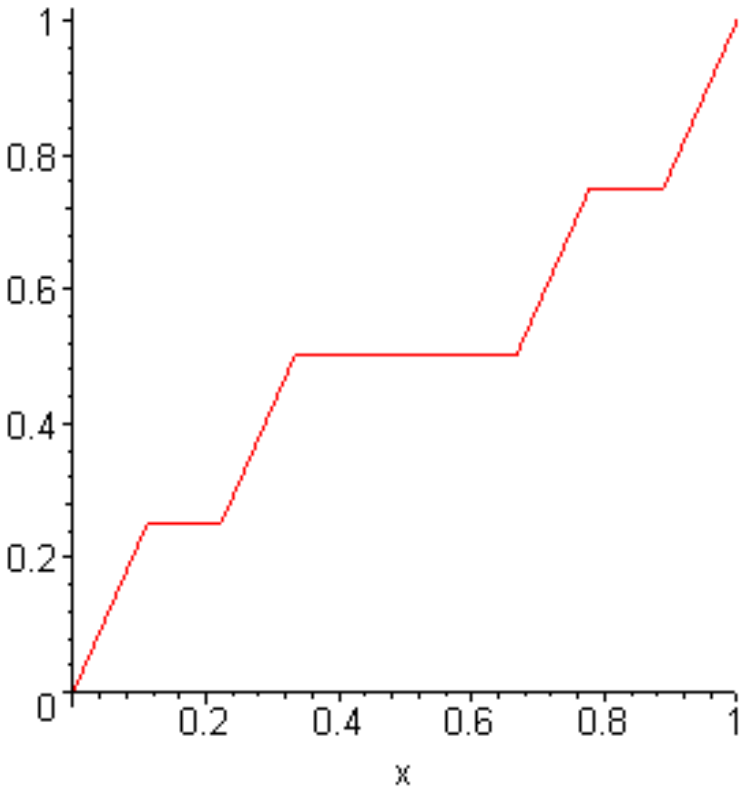


As we can see, the cantor set occupies less and less space on the interval [0,1], as advertised.

Our next topic of interest is the Cantor Function. This function is obtained from a limit of a sequence of functions directly obtained from the construction of the Cantor Set. The construction is as follows, we start with the first step of the construction of the Cantor Set where we have removed the middle third of the interval [0,1]. We define our first function to be 1/2 on that segment removed, (1/3,2/3). We define this function to be 0 at **x**=0, and 1 at **x**=1. For the rest of the values, we simple make the straight line connections, so that our function looks like:



This is the first function in our sequence. For the next function, we consider all of the segments removed up to the second step of the Cantor Set construction, which are (1/9,2/9),(1/3,2/3), and (7/9,8/9). We define our function to be 1/4 on the first segment, 1/2 on the second, and 3/4 on the third. We let it be 0 at 0 and 1 at 1, and make the straight line connections everywhere else, so that our function looks like:



We then proceed as such. What we will get is a sequence of uniformly convergent continuous functions that must converge to a continuous function. We call this limiting function the Cantor Function, which has very interesting properties. We have constructed our sequence of functions so that at the **k**th step of the construction of the Cantor Set, all of our functions after this step are all fixed constants on each segment removed up to the **k**th step of the construction. Thus, our limiting function will be constant on all of the segments removed in the construction of the Cantor Set, which has measure 1. On each of these segments, the Cantor Function is differentiable and has derivative zero. Thus, the Cantor Function is differentiable almost everywhere on [0,1] with derivative zero. However, the Cantor Function is a limit of increasing functions, and so must be non-decreasing. In fact, each function in our sequence leading up to the Cantor Function has value 0 at 0 and 1 at 1, thus the value of the Cantor Function is 0 at 0 and 1 at 1. Thus, we have managed to produce a function that is differentiable almost everywhere on [0,1] with derivative 0, and yet it manages to climb up from 0 all the way up to one! As Jason Lee remarked, if you blink on a set of measure zero, the Cantor Function is up there.

Let us now write a program which gives us our sequence of functions which lead up to the Cantor Function. The main obstacle is the same as the one in the program for the **ComplementFunction**, which is that our sequence of functions are defined piecewise but how many pieces and where they start and end changes at each step. However, we use the same technique of using sums here as was used in **ComplementFunction.**

Thus, firstly we make a function which accounts for the places where our sequence of functions are constant:

> **`cfsteps:=proc(x,i,k)`**

```
if cantorpoint(2*i,k) < x and x <= cantorpoint(2*i+1,k)
then i/(2^k) else 0
fi end ;
```

cfsteps := **proc** (*x*, *i*, *k*)
  if cantorpoint(2×*i*, *k*) < *x* **and** *x* ≤ cantorpoint(2×*i* + 1, *k*) **then** *i*/2^*k*
  else 0
  end if
end proc

for each **k**, **cfsteps** draws each individual piece of our **k**th function where it is constant. To get a function that is constant on all of the appropriate segements, we simply do as before and take a sum:

```
> cfallsteps:= (x,k) -> sum('cfsteps(x,i,k)', 'i'=1..((2^k)-
1));
```

$$cfallsteps := (x, k) \to \sum_{'i'=1}^{2^k-1} \text{'cfsteps}(x, i, k)\text{'}$$

Now, for a fixed **k**, we must make all of the straight line connections. Again, we make each piece seperately. Since each piece is a line, we can readibly come up with a formula based on the cantor points. First we make some preliminary definitions:

```
> cflinepart1:= (x,i,k) -> ((3^k)/((2^k)(cantorpoint(2*i,k)-
cantorpoint(2*i-1,k))))*x
;
```

$$cflinepart1 := (x, i, k) \to \frac{3^k x}{(2^k)(\text{cantorpoint}(2\,i, k) - \text{cantorpoint}(2\,i - 1, k))}$$

```
> cflinepart2:= (i,k) ->
(3^k)*(((i-1)*cantorpoint(2*i,k)-i*cantorpoint(2*i-
1,k))/((2^k)(cantorpoint(2*i,k)-cantorpoint(2*i-1,k))));
```

$$cflinepart2 := (i, k) \to \frac{3^k ((i - 1)\,\text{cantorpoint}(2\,i, k) - i\,\text{cantorpoint}(2\,i - 1, k))}{(2^k)(\text{cantorpoint}(2\,i, k) - \text{cantorpoint}(2\,i - 1, k))}$$

and now we define each line:

```
> cfline:=proc(x,i,k)
if cantorpoint(2*i-1,k) < x and x <= cantorpoint(2*i,k)
then cflinepart1(x,i,k)+cflinepart2(i,k) else  0
fi end ;
```

cfline := **proc** (*x*, *i*, *k*)
  if cantorpoint(2×*i* − 1, *k*) < *x* **and** *x* ≤ cantorpoint(2×*i*, *k*) **then**
    cflinepart1(*x*, *i*, *k*) + cflinepart2(*i*, *k*)
  else 0
  end if
end proc

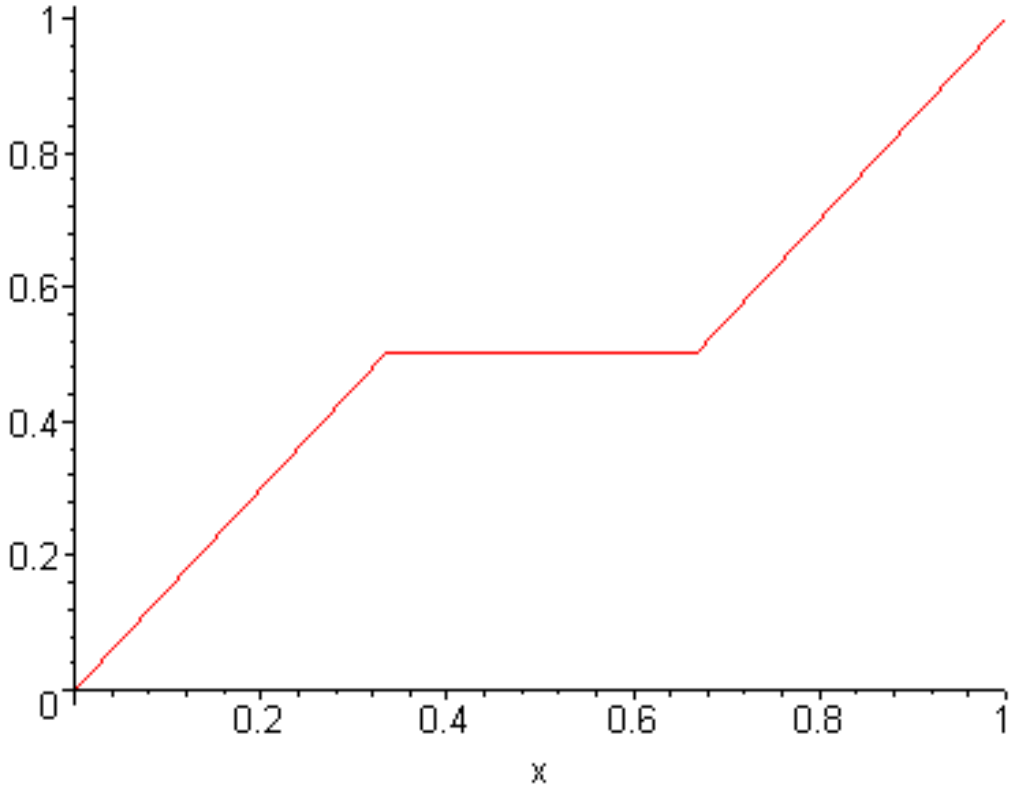
We are now ready to define our sequence. We define **CantorFunction(x,k)** to be our sequence of functions, and we use the same trick of using sums:

```
> CantorFunction:=(x,k)-
>sum('cfline(x,i,k)','i'=1..(2^(k)))+'cfallsteps(x,k)' ;
```

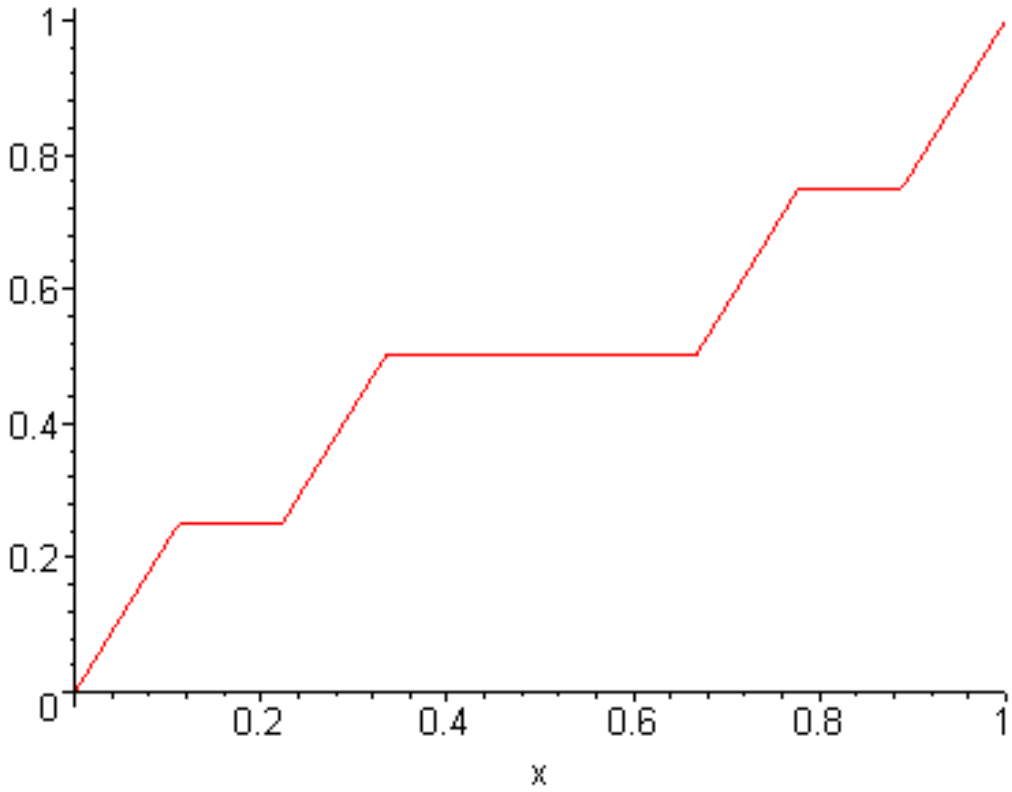
$$CantorFunction := (x, k) \rightarrow \left(\sum_{'i'=1}^{2^k} \text{'cfline}(x, i, k)' \right) + \text{'cfallsteps}(x, k)'$$

Let us plot this for several **k**:

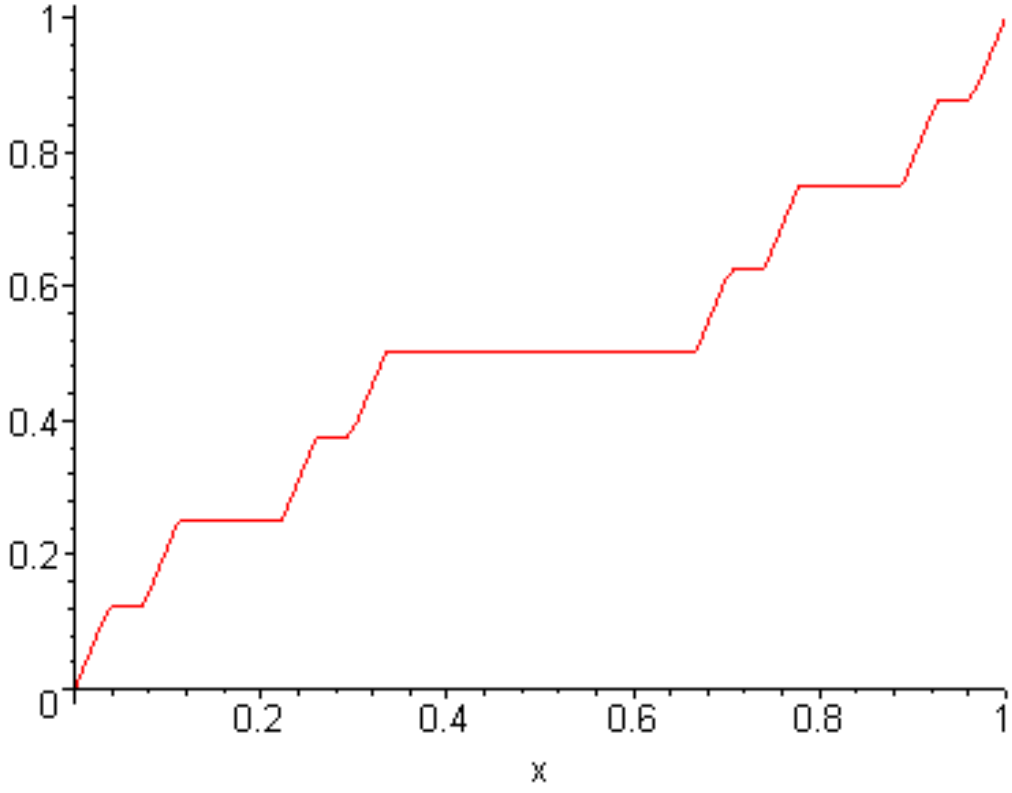
```
> plot('CantorFunction(x,1)',x=0..1);
```



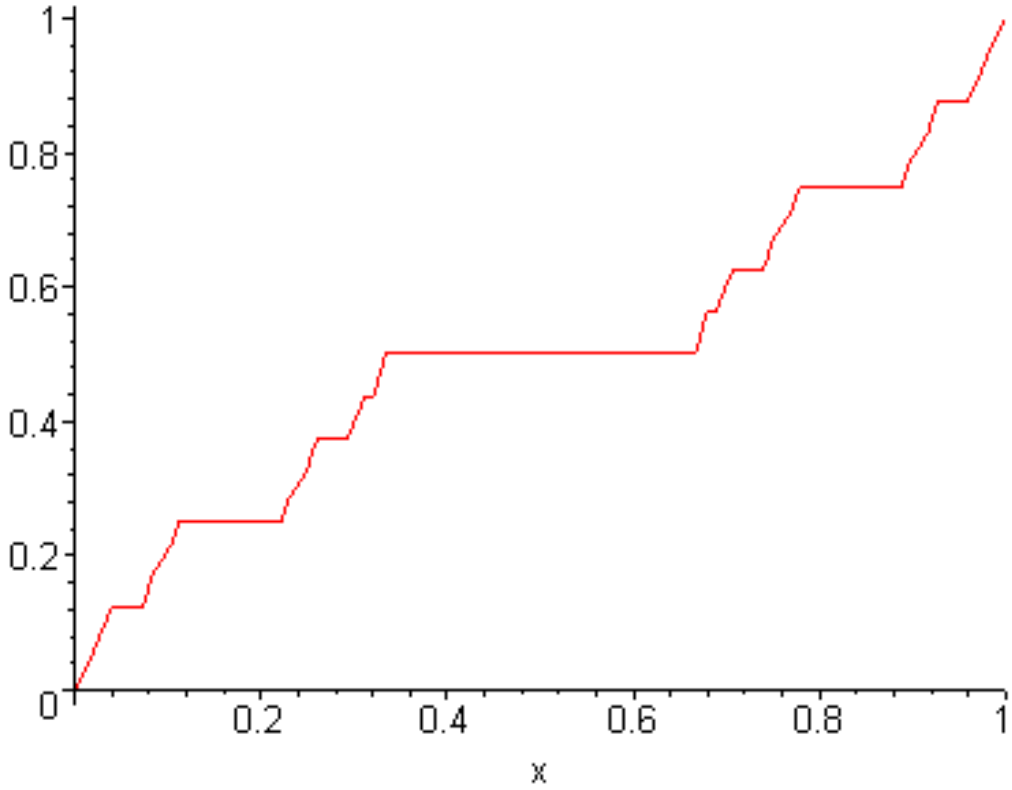

```
> plot('CantorFunction(x,2)',x=0..1);
```



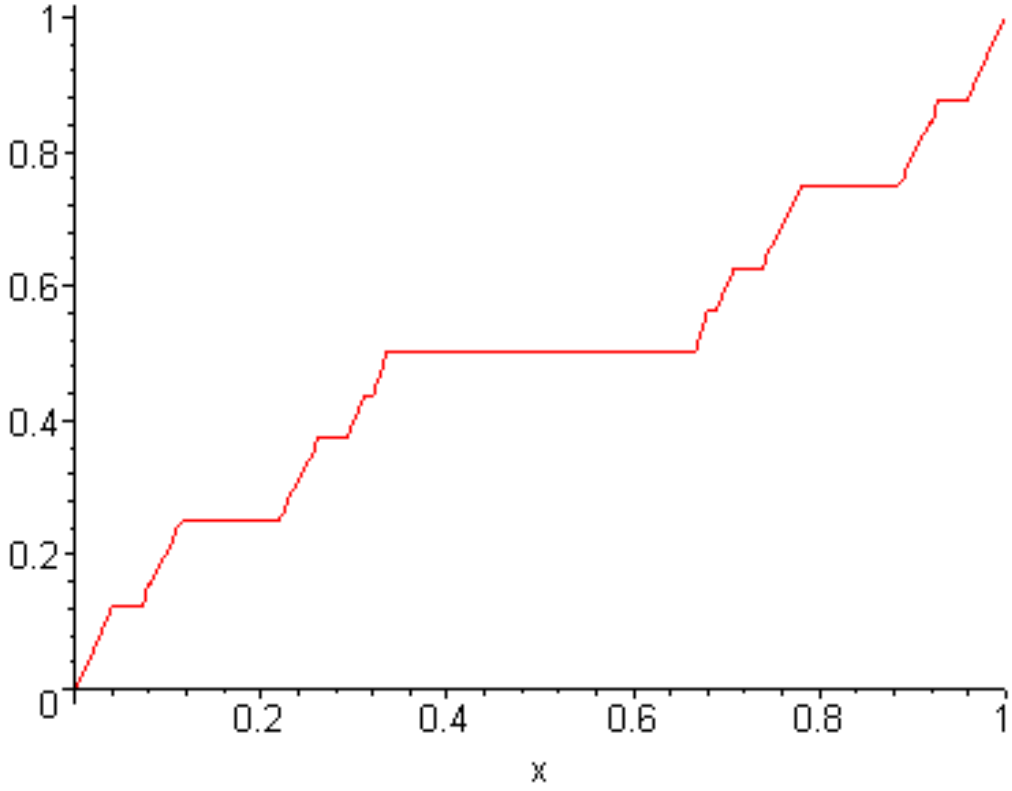

```
> plot('CantorFunction(x,3)',x=0..1);
```



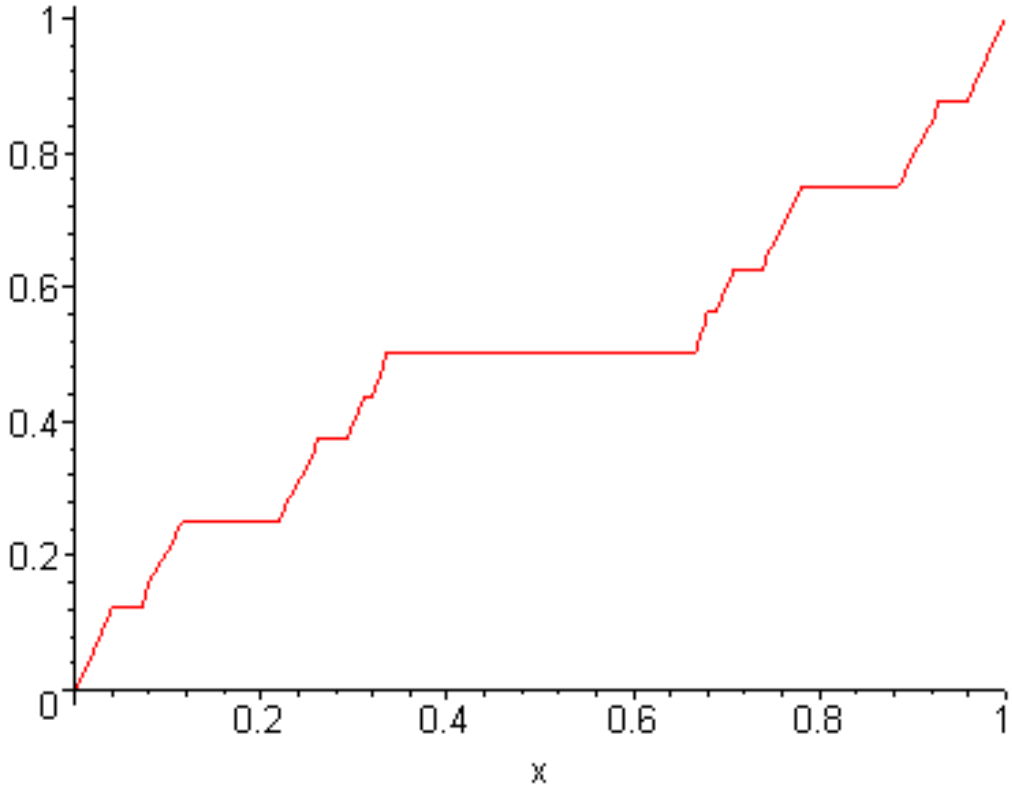

```
> plot('CantorFunction(x,4)',x=0..1);
```



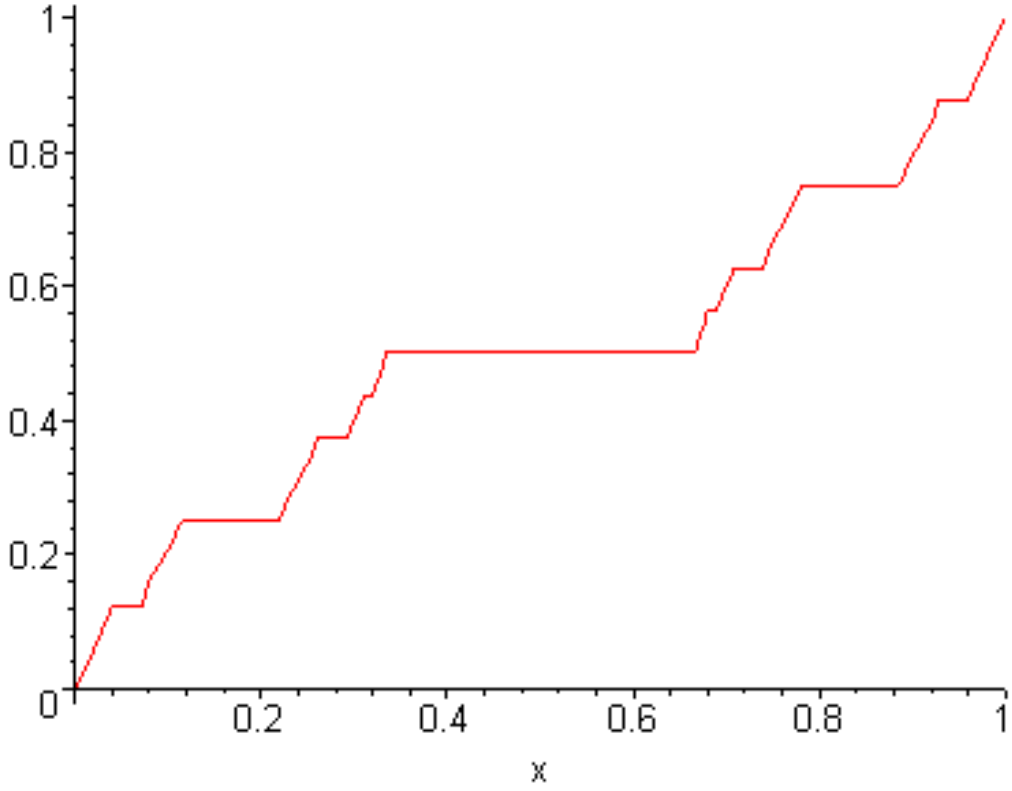

```
> plot('CantorFunction(x,5)',x=0..1);
```



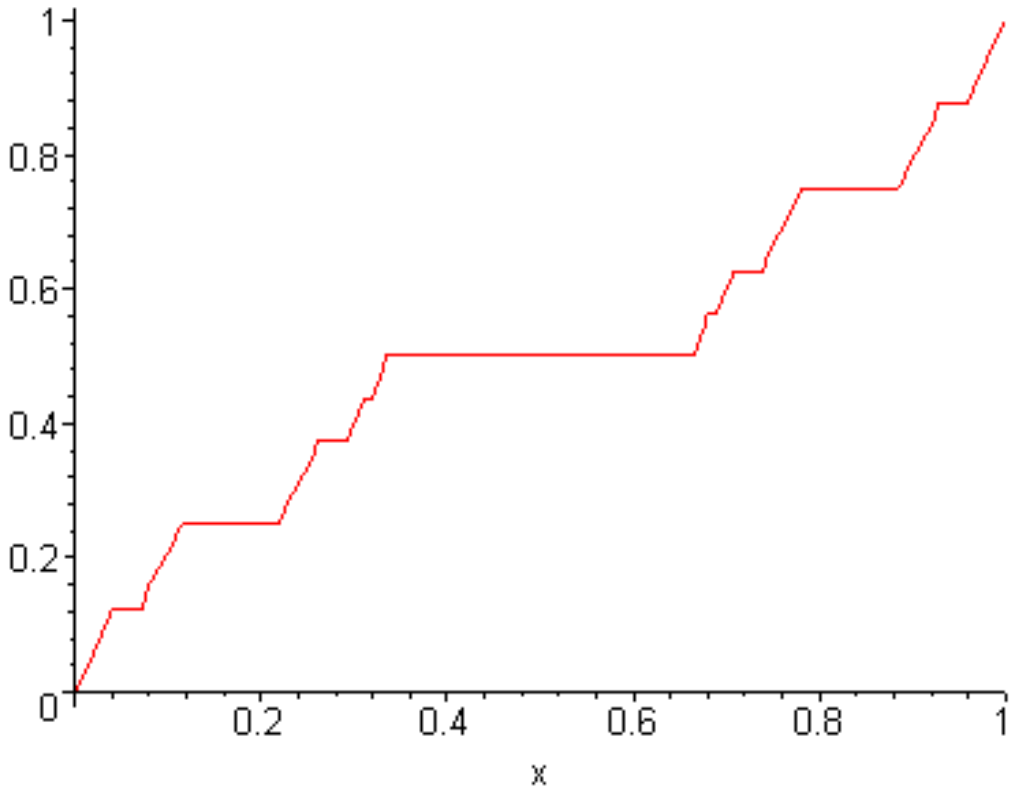

```
> plot('CantorFunction(x,6)',x=0..1);
```




```
> plot('CantorFunction(x,7)',x=0..1);
```




```
> plot('CantorFunction(x,8)',x=0..1);
```



the Cantor Function is also referred to as the devil's staircase, and we can see why. Graphically we can also be convinced that these curves are converging uniformly. Indeed, each successive curve will be constant where the previous one was, so that the maximum difference between one function in our sequence and another after it must occur somewhere in between the places where the first curve is constant. As such, then it is only a matter of getting our straight line connections to be sufficiently close to each other, which pictorially we can see is not such a far-fetched thing to do.

One can now contemplate what would happen if instead of removing the middle thirds in the construction of the Cantor Set, we remove some other length strictly between 0 and 1. If we do this, we get what is called the General Cantor Set. These sets have interesing properties as well. They are all uncountable, perfect, compact, and are totally disconnected. However, it is possible to construct a General Cantor Set that has positive measure if we allow how much we remove at each step change. Moreover, given any number strictly between 0 and 1, we can produce a general cantor set that has that value as its measure. Let us investigate the General Cantor Set.

General Cantor Set

Our task will be to produce an algorithm which gives us the points in the construction of the General Cantor Set. More specifically, given an **i,k** and a **b**, this algorithm will give us the **i**th point used in the **k**th step of the construction of the cantor set according to the rule that we remove the middle **b** of each segment. For now, we regard the **b** as fixed at every step, and **b** is always strictly between 0 and 1.

This algorithm will be a doubly recursive one. We will first manually produce the four points that are used in the first step of the general construction, which will directly depend on **b**. Then we will build on these points to get all the other points for every step beyond the first step. As such, the first set of points is given by:

```
> firstgencantset:= proc(b) [0,(1/2)*(1-b),(1/2)*(1+b),1]
end;
```

$$firstgencantset := \mathbf{proc}(b)\ [\,0,\ -1/2\times b + 1/2,\ 1/2 + 1/2\times b,\ 1\,]\ \mathbf{end\ proc}$$

one can check that the segment ((1/2)*(1-**b**),(1/2)*(1+**b**)) has length **b**. Now we define a preliminary function, which we also used in the original Cantor Set, but that we describe explicitly here for simplicity.

```
> factorsoftwo:= (i)-> ifactors(i)[2][1][2];
```

$$factorsoftwo := i \rightarrow \mathrm{ifactors}(i)_{2}{}_{1}{}_{2}$$

Now we define our points:

```
> gencantpoint:=proc(i,k,b)
if k=0 then 1 ; else
if k=1 then (firstgencantset(b))[i] ; else
if i=1 then 0 else
if i mod 2 = 0 then
if factorsoftwo(i)=1 then gencantpoint(i-1,k,b)+(((1/2)*(1-
b))^k) ;
else gencantpoint(i*(2^(1-factorsoftwo(i))),k+1-
factorsoftwo(i),b) ; fi ;
else if factorsoftwo(i-1)=1 then gencantpoint(i-
1,k,b)+b*(((1/2)*(1-b))^(k-1)) ;
else gencantpoint((i-1)*(2^(1-factorsoftwo(i-1)))+1,k+1-
factorsoftwo(i-1),b) ;
fi ; fi ; fi ; fi ; fi ; end ;
```

$$gencantpoint := \mathbf{proc}(i, k, b)$$
$$\quad \mathbf{if}\ k = 0\ \mathbf{then}\ 1$$
$$\quad \mathbf{else}$$
$$\quad\quad \mathbf{if}\ k = 1\ \mathbf{then}\ \mathrm{firstgencantset}(b)[\,i\,]$$
$$\quad\quad \mathbf{else}$$
$$\quad\quad\quad \mathbf{if}\ i = 1\ \mathbf{then}\ 0$$
$$\quad\quad\quad \mathbf{else}$$
$$\quad\quad\quad\quad \mathbf{if}\ i \bmod 2 = 0\ \mathbf{then}$$
$$\quad\quad\quad\quad\quad \mathbf{if}\ \mathrm{factorsoftwo}(i) = 1\ \mathbf{then}$$
$$\quad\quad\quad\quad\quad\quad \mathrm{gencantpoint}(i - 1, k, b) + (1/2 - 1/2\times b)\,\hat{}\,k$$

```
                else gencantpoint( i×2^( 1 – factorsoftwo( i ) ),
                     k + 1 – factorsoftwo( i ), b )
                end if
            else
                if factorsoftwo( i – 1 ) = 1 then
                     gencantpoint( i – 1, k, b ) + b×( 1/2 – 1/2×b )^( k – 1 )
                else gencantpoint( ( i – 1 )×2^( 1 – factorsoftwo( i – 1 ) ) + 1,
                     k + 1 – factorsoftwo( i – 1 ), b )
                end if
            end if
        end if
    end if
end proc
```

This is a formidable algorithm, one that uses a heavy recursion. In order to get the **i**th point used in the **k**th step, this algorithm not only investigates the the nature of the **i**, but it also looks at the divisibility of **i**-1 by 2, and then draws on the points used in the previous step of the construction.

Now we define as before, a procedure which lists out the sets that are left over at each step in our construction:

```
> gencantset:=proc(k,b) [seq([gencantpoint(2*i-
1,k,b),gencantpoint(2*i,k,b)],i=1..(2^k))] end;
```

```
gencantset := proc (k, b)
    [ seq( [ gencantpoint( 2×i – 1, k, b ), gencantpoint( 2×i, k, b ) ], i = 1 .. 2^k ) ]
end proc
```

Now we can check that algorithm is correct, by putting in the value of **b**=1/3. This should give us the regular Cantor Set, and we list **gencantset** and **cantorset** for several values of **k** to verify this:

```
> gencantset(1,1/3);
```

$$\left[\left[0, \frac{1}{3}\right], \left[\frac{2}{3}, 1\right]\right]$$

```
> cantorset(1,1/3);
```

$$\left[\left[0, \frac{1}{3}\right], \left[\frac{2}{3}, 1\right]\right]$$

```
> gencantset(2,1/3);
```

$$\left[\left[0, \frac{1}{9}\right], \left[\frac{2}{9}, \frac{1}{3}\right], \left[\frac{2}{3}, \frac{7}{9}\right], \left[\frac{8}{9}, 1\right]\right]$$

```
> cantorset(2);
```

$$\left[\left[0, \frac{1}{9}\right], \left[\frac{2}{9}, \frac{1}{3}\right], \left[\frac{2}{3}, \frac{7}{9}\right], \left[\frac{8}{9}, 1\right]\right]$$

```
> gencantset(3,1/3);
```

$$\left[\left[0, \frac{1}{27}\right], \left[\frac{2}{27}, \frac{1}{9}\right], \left[\frac{2}{9}, \frac{7}{27}\right], \left[\frac{8}{27}, \frac{1}{3}\right], \left[\frac{2}{3}, \frac{19}{27}\right], \left[\frac{20}{27}, \frac{7}{9}\right], \left[\frac{8}{9}, \frac{25}{27}\right], \left[\frac{26}{27}, 1\right]\right]$$

```
> cantorset(3);
```

$$\left[\left[0, \frac{1}{27}\right], \left[\frac{2}{27}, \frac{1}{9}\right], \left[\frac{2}{9}, \frac{7}{27}\right], \left[\frac{8}{27}, \frac{1}{3}\right], \left[\frac{2}{3}, \frac{19}{27}\right], \left[\frac{20}{27}, \frac{7}{9}\right], \left[\frac{8}{9}, \frac{25}{27}\right], \left[\frac{26}{27}, 1\right]\right]$$

```
> gencantset(4,1/3);
```

$$\left[\left[0, \frac{1}{81}\right], \left[\frac{2}{81}, \frac{1}{27}\right], \left[\frac{2}{27}, \frac{7}{81}\right], \left[\frac{8}{81}, \frac{1}{9}\right], \left[\frac{2}{9}, \frac{19}{81}\right], \left[\frac{20}{81}, \frac{7}{27}\right], \left[\frac{8}{27}, \frac{25}{81}\right], \left[\frac{26}{81}, \frac{1}{3}\right], \left[\frac{2}{3}, \frac{55}{81}\right], \left[\frac{56}{81}, \frac{19}{27}\right], \left[\frac{20}{27}, \frac{61}{81}\right], \left[\frac{62}{81}, \frac{7}{9}\right], \left[\frac{8}{9}, \frac{73}{81}\right], \left[\frac{74}{81}, \frac{25}{27}\right], \left[\frac{26}{27}, \frac{79}{81}\right], \left[\frac{80}{81}, 1\right]\right]$$

```
> cantorset(4);
```

$$\left[\left[0, \frac{1}{81}\right], \left[\frac{2}{81}, \frac{1}{27}\right], \left[\frac{2}{27}, \frac{7}{81}\right], \left[\frac{8}{81}, \frac{1}{9}\right], \left[\frac{2}{9}, \frac{19}{81}\right], \left[\frac{20}{81}, \frac{7}{27}\right], \left[\frac{8}{27}, \frac{25}{81}\right], \left[\frac{26}{81}, \frac{1}{3}\right], \left[\frac{2}{3}, \frac{55}{81}\right], \left[\frac{56}{81}, \frac{19}{27}\right], \left[\frac{20}{27}, \frac{61}{81}\right], \left[\frac{62}{81}, \frac{7}{9}\right], \left[\frac{8}{9}, \frac{73}{81}\right], \left[\frac{74}{81}, \frac{25}{27}\right], \left[\frac{26}{27}, \frac{79}{81}\right], \left[\frac{80}{81}, 1\right]\right]$$

It seems to check out. Now out of curiosity, let us put in **b**=1/2 and list some of the steps:

```
> gencantset(1,1/2);
```

$$\left[\left[0, \frac{1}{4}\right], \left[\frac{3}{4}, 1\right]\right]$$

```
> gencantset(2,1/2);
```

$$\left[\left[0, \frac{1}{16}\right], \left[\frac{3}{16}, \frac{1}{4}\right], \left[\frac{3}{4}, \frac{13}{16}\right], \left[\frac{15}{16}, 1\right]\right]$$

```
> gencantset(3,1/2);
```

$$\left[\left[0, \frac{1}{64}\right], \left[\frac{3}{64}, \frac{1}{16}\right], \left[\frac{3}{16}, \frac{13}{64}\right], \left[\frac{15}{64}, \frac{1}{4}\right], \left[\frac{3}{4}, \frac{49}{64}\right], \left[\frac{51}{64}, \frac{13}{16}\right], \left[\frac{15}{16}, \frac{61}{64}\right], \left[\frac{63}{64}, 1\right]\right]$$

```
> gencantset(4,1/2);
```

$$\left[\left[0, \frac{1}{256}\right], \left[\frac{3}{256}, \frac{1}{64}\right], \left[\frac{3}{64}, \frac{13}{256}\right], \left[\frac{15}{256}, \frac{1}{16}\right], \left[\frac{3}{16}, \frac{49}{256}\right], \left[\frac{51}{256}, \frac{13}{64}\right], \left[\frac{15}{64}, \frac{61}{256}\right], \left[\frac{63}{256}, \frac{1}{4}\right], \left[\frac{3}{4}, \frac{193}{256}\right], \left[\frac{195}{256}, \frac{49}{64}\right], \left[\frac{51}{64}, \frac{205}{256}\right], \left[\frac{207}{256}, \frac{13}{16}\right], \left[\frac{15}{16}, \frac{241}{256}\right], \left[\frac{243}{256}, \frac{61}{64}\right], \left[\frac{63}{64}, \frac{253}{256}\right], \left[\frac{255}{256}, 1\right]\right]$$

Now, let us measure take the measure the intervals left in each step of this construction:

```
> measure(gencantset(1,1/2));
```

$$\frac{1}{2}$$

```
> measure(gencantset(2,1/2));
```

$$\frac{1}{4}$$

```
> measure(gencantset(3,1/2));
```

$$\frac{1}{8}$$

```
> measure(gencantset(4,1/2));
```

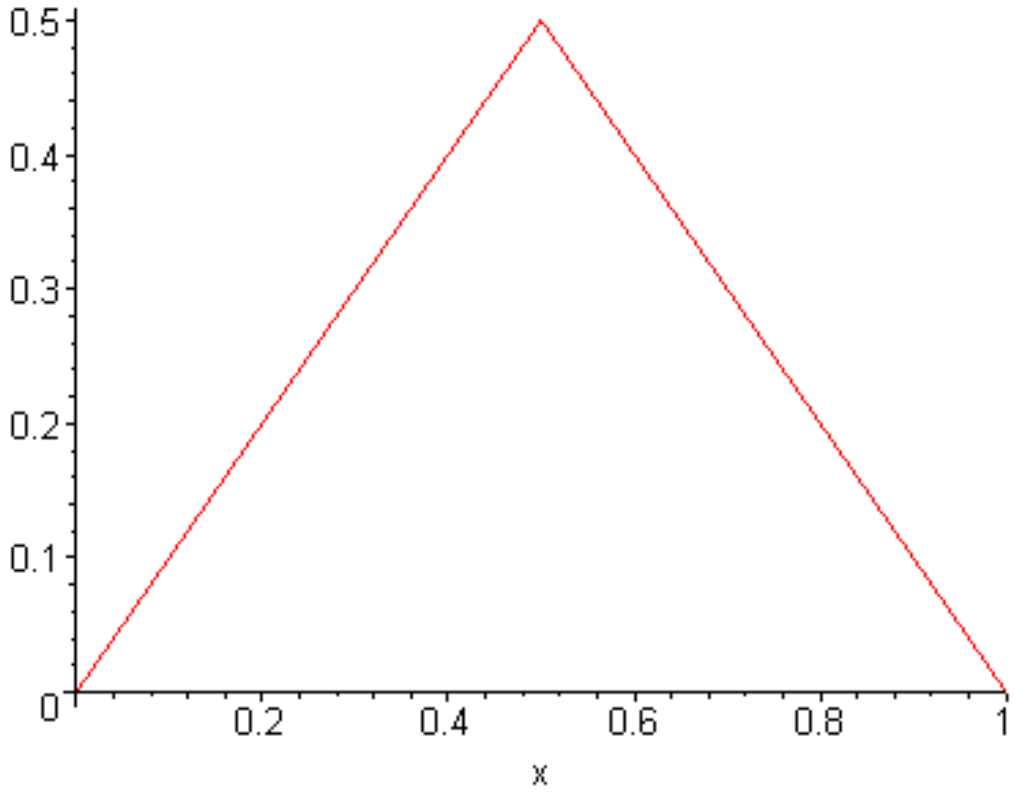
$$\frac{1}{16}$$

We see that the measure of this cantor set will be zero. Indeed, any general cantor set will have measure zero when **b** is fixed. This is again more apparent when we investigate the measure of the compleement of such a general cantor set on [0,1]. The length of the first interval removed is **b**. We are left with two intervals of length (1/2)*(1-**b**). At the second step, we remove two intervals each of length **b***(1/2)*(1-**b**). All of the segments removed are disjoint. As such, at the kth step, we remove 2^**k** segments of length **b***((1/2)*(1-**b**))^**k**. All of these segements will be disjoint, so to get the measure of the complement of this Cantor Set, we sum the lengths of each of the pieces. Including the first step, we get the sum from 0 to infinity of (2^k)***b***((1/2)*(1-**b**))^**k**. The summand simplifies to **b***(1-**b**)^**k**. This is again a geometric series, which will sum to 1.

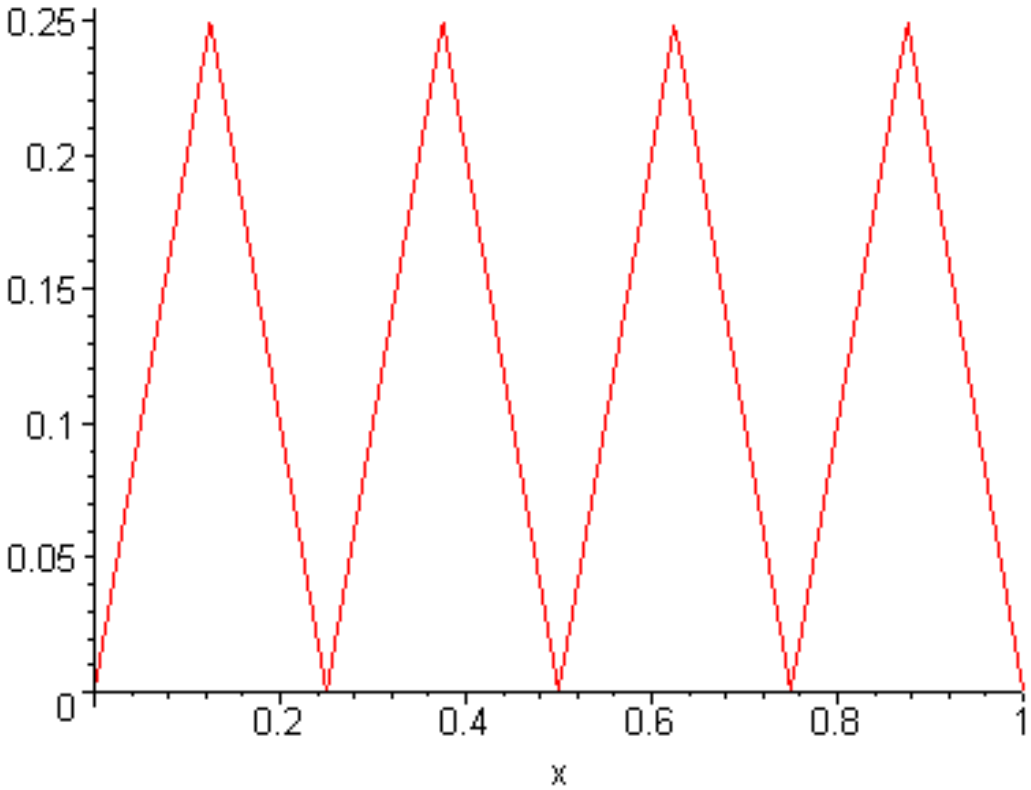
At this point one can continue as in the Cantor Set and define a general Devil's Staircase based on the General Cantor Set.
Our next example deals with a classic question which comes to the mind of most undergraduate mathematics students. Is every continuous function differentiable? The answer is no, since the absolute value function is continuous at zero but not differentiable there. However, we can ask, is there a function which is continuous everywhere but differentiable nowhere? The answer is in the affirmative, and we now explore an example of such a function.

Van der Waerden Nowhere Differentiable Function

Our example of a continuous nowhere differentiable function is due to Bartel Leendert van der Waerden, and it will be obtained from taking the sum of a sequence of functions. This sequence of functions is best described through graphs. The first function is a simple sawtooth:



the second function is another saw tooth, with four teeth:



However, the height of each sawtooth is reduced by a half to 1/4. We proceed as such, at each **k**th step getting a function that has 2^**k** sawteeth, each of height 1/2^**k**. The sum of

these functions is uniformly convergent by the Weierstrauss M-test, since the **k**th function is bounded by 1/2^**k**. Thus, we call the resulting sum the Van der Waerden Function, and it is continuous since it is the uniformly convergent sum of a sequence of continuous functions. However, it fails to be differentiable anywhere on [0,1]. I shall give some reasoning why later.

Now let us write a program that gives us the **n**th partial sum of our sequence of sawtooth functions. First, we must get this sequence of functions. This task is easy enough given our strategies. We first define each of the pieces of each sawtooth, and then sum them all in the end:

```
> nowherediffeven:= proc(x,i,k)
if 2*i/(2^(k+1)) <= x and x < (2*i+1)/(2^(k+1))
then 2*x-(2*i/(2^k)) else 0
fi end ;
```

nowherediffeven := **proc** (*x*, *i*, *k*)
  if 2×*i*/2^(*k* + 1) ≤ *x* **and** *x* < (2×*i* + 1)/2^(*k* + 1) **then** 2×*x* – 2×*i*/2^*k*
  else 0
  end if
end proc

```
> nowherediffodd:= proc(x,i,k)
if (2*i+1)/(2^(k+1)) <= x and x < (2*i+2)/(2^(k+1))
then -2*x+((2*i+2)/(2^k)) else 0
fi end ;
```

nowherediffodd := **proc** (*x*, *i*, *k*)
  if (2×*i* + 1)/2^(*k* + 1) ≤ *x* **and** *x* < (2×*i* + 2)/2^(*k* + 1) **then**
    –2×*x* + (2×*i* + 2)/2^*k*
  else 0
  end if
end proc

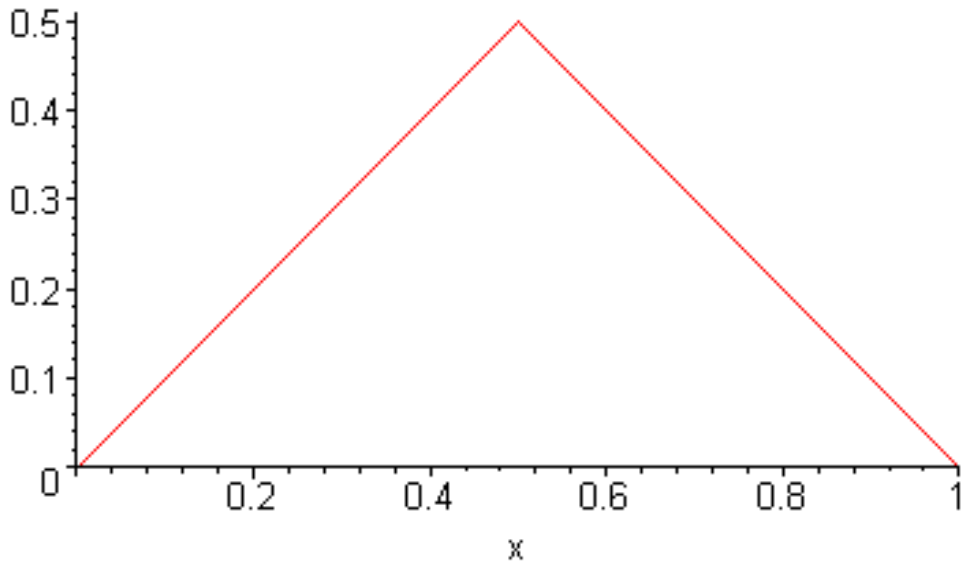
These describe the left and right pieces of each sawtooth. Now, we add all the pieces up using our sum technique:

```
> nowherediffseq:= proc(x,k)
if k=1
then if x <= 1/2 then x else -x+1 fi
else sum('nowherediffeven(x,i,k)', i=0...(2^k)-
1)+sum('nowherediffodd(x,i,k)', i=0...(2^k)-1)
fi end ;
```

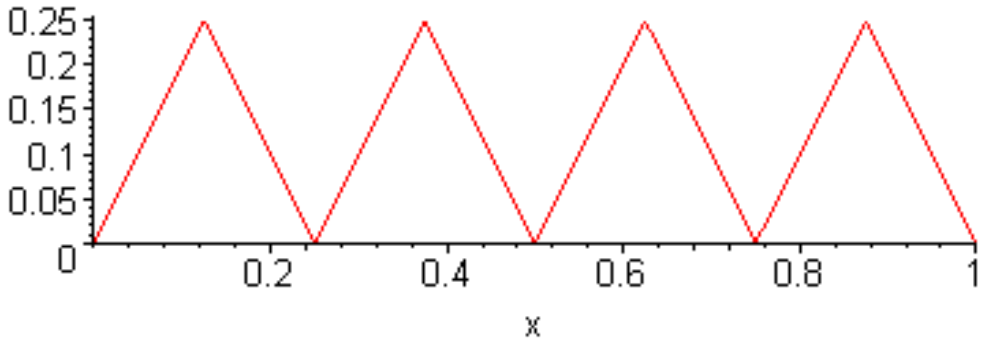
nowherediffseq := **proc** (*x*, *k*)
  if *k* = 1 **then** **if** *x* ≤ 1/2 **then** *x* **else** –*x* + 1 **end if**
  else sum('nowherediffeven(*x*, *i*, *k*)', *i* = 0 .. 2^*k* – 1)
    + sum('nowherediffodd(*x*, *i*, *k*)', *i* = 0 .. 2^*k* – 1)
  end if
end proc

Let us plot our sequence:

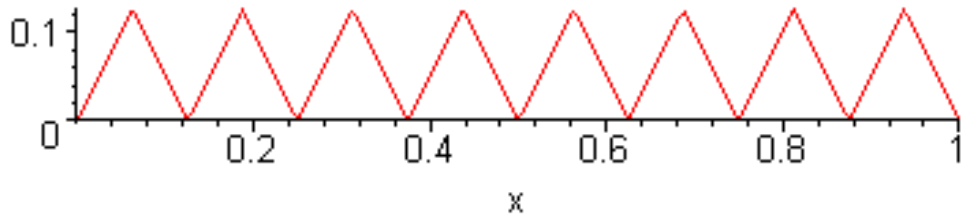
```
> plot('nowherediffseq(x,1)',x=0..1);
```



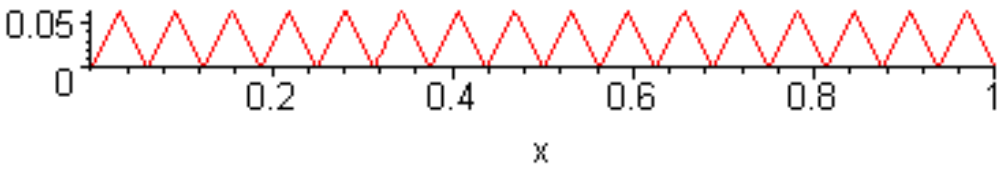

```
> plot('nowherediffseq(x,2)',x=0..1);
```



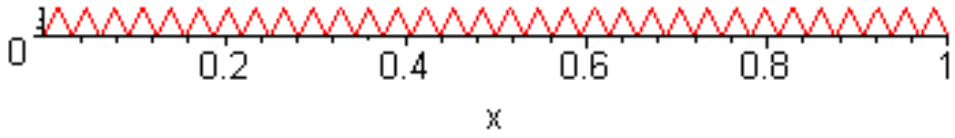

```
> plot('nowherediffseq(x,3)',x=0..1);
```




```
> plot('nowherediffseq(x,4)',x=0..1);
```




```
> plot('nowherediffseq(x,5)',x=0..1);
```



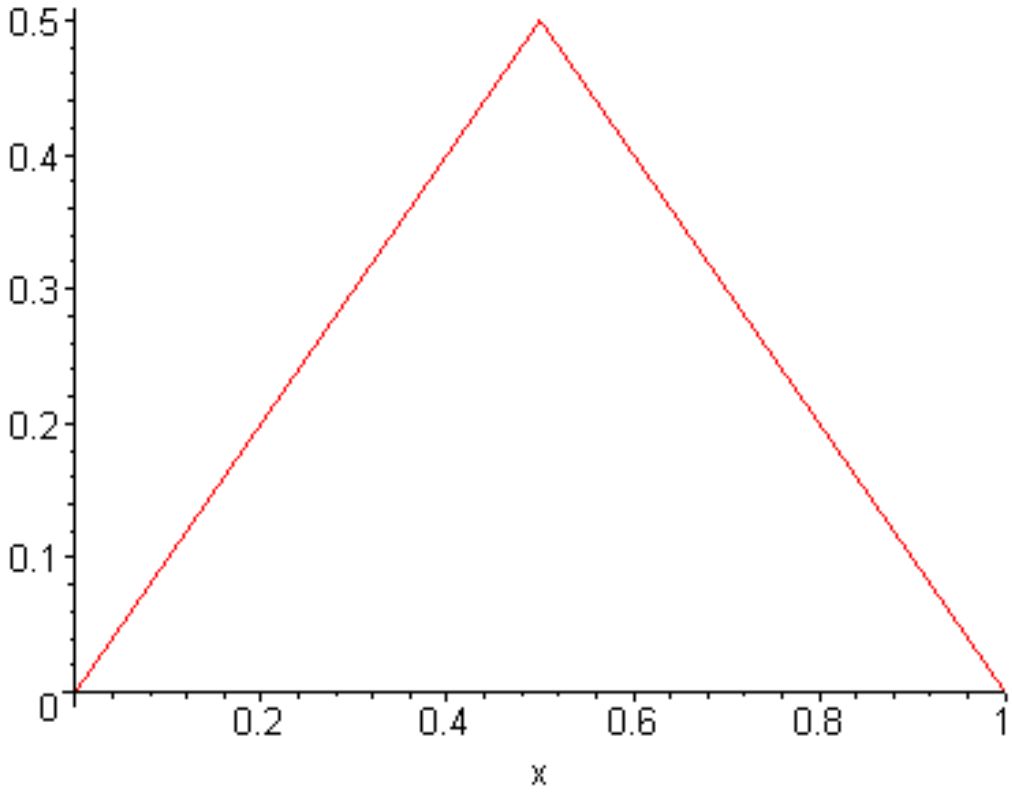
As we can see, each successive function has more peaks. We now define the **n**th partial sum of these functions, which shall approximate the Van der Waerden Function:

```
> nowherediff:=(x,N)-> sum('nowherediffseq(x,k)',k=1..N);
```

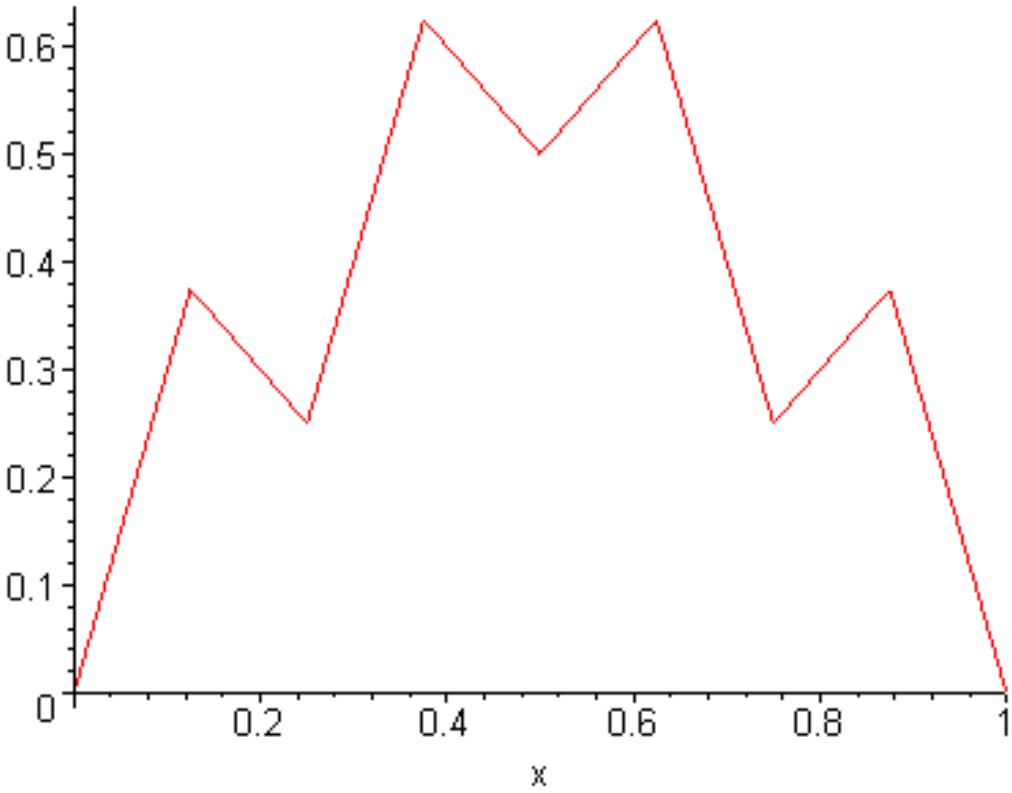
$$nowherediff := (x, N) \rightarrow \sum_{k=1}^{N} \text{'nowherediffseq}(x, k)\text{'}$$

Let us plot this function:

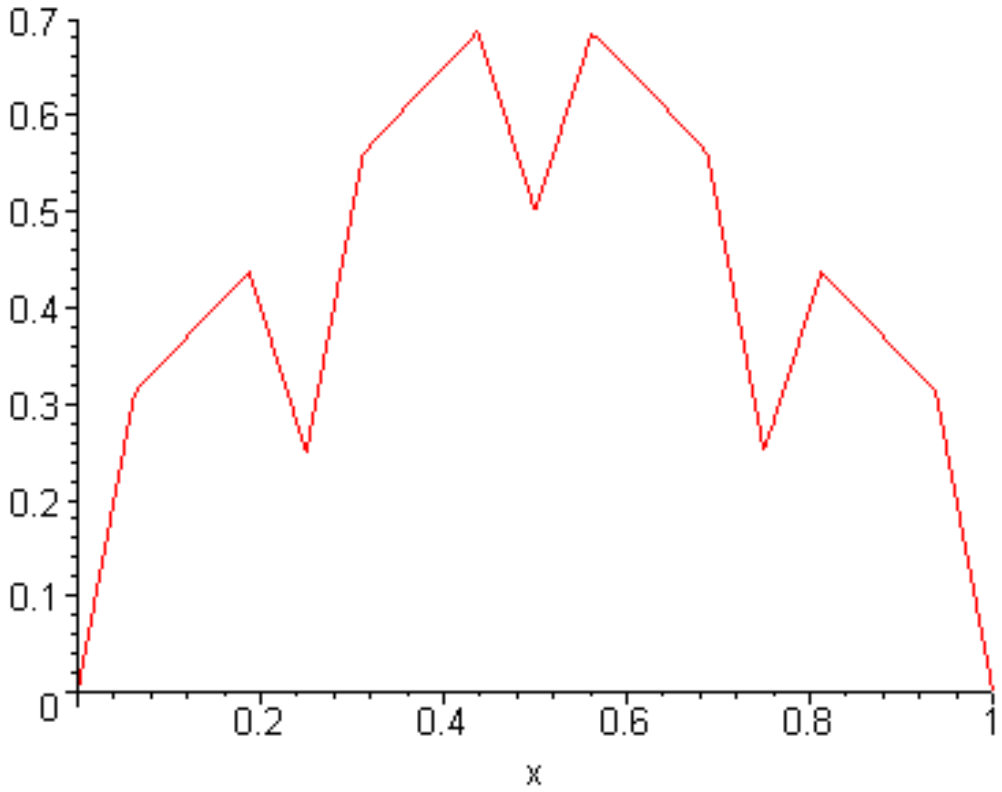
```
> plot('nowherediff(x,1)',x=0..1);
```



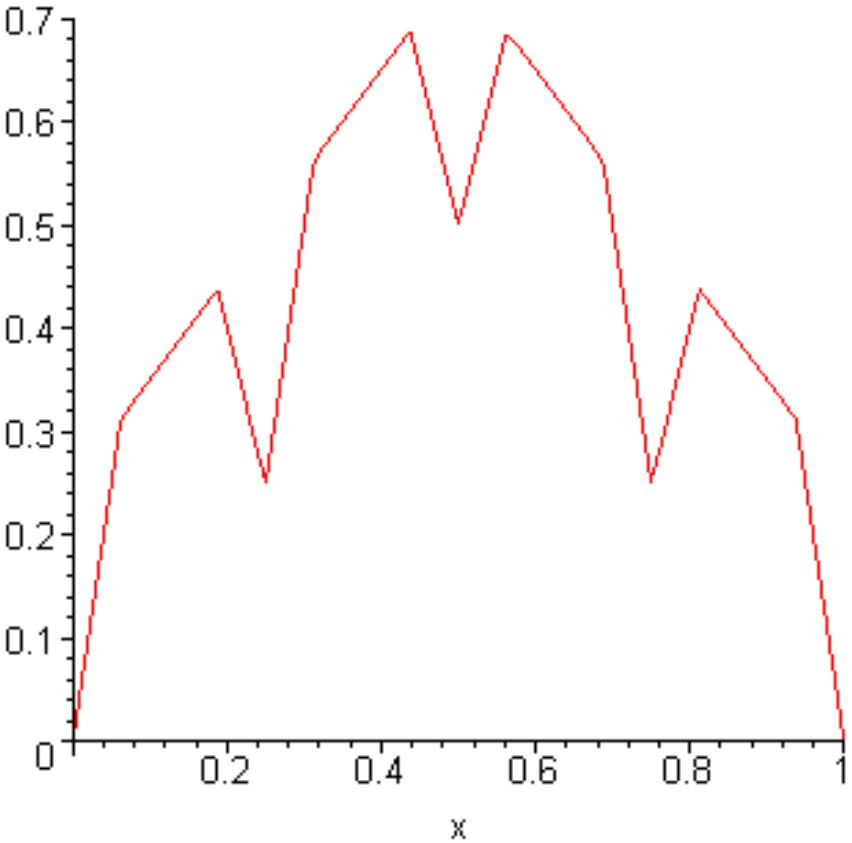

```
> plot('nowherediff(x,2)',x=0..1);
```



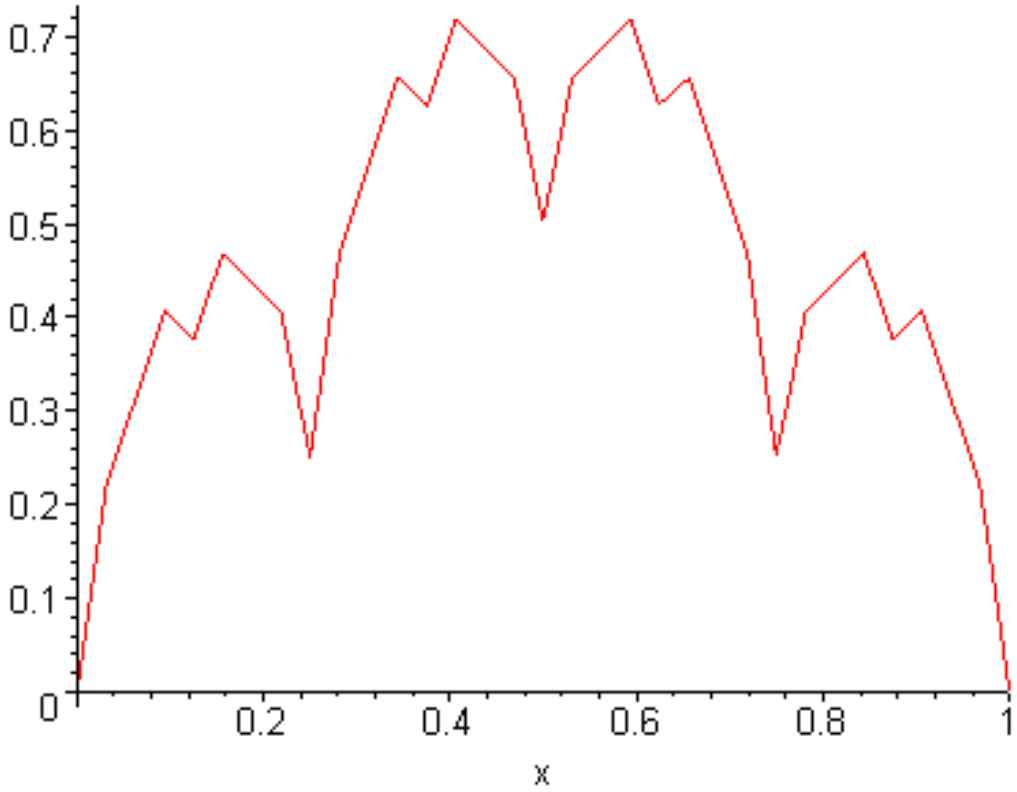

```
> plot('nowherediff(x,3)',x=0..1);
```



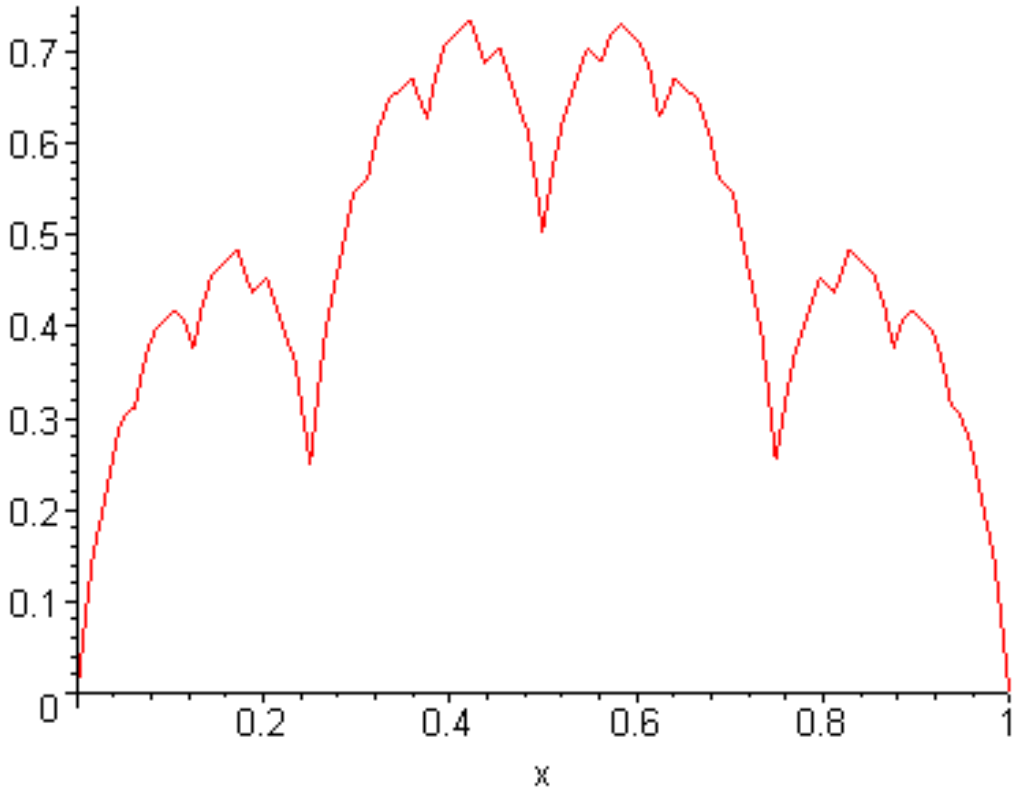

```
> plot('nowherediff(x,3)',x=0..1);
```



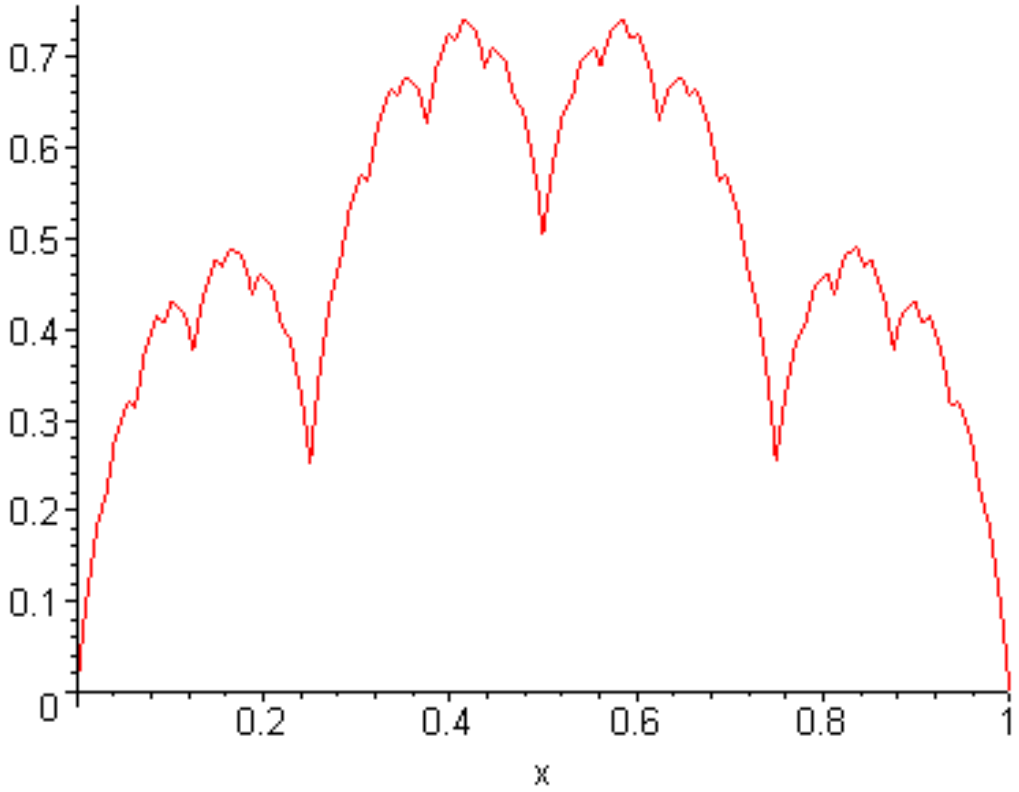

```
> plot('nowherediff(x,4)',x=0..1);
```



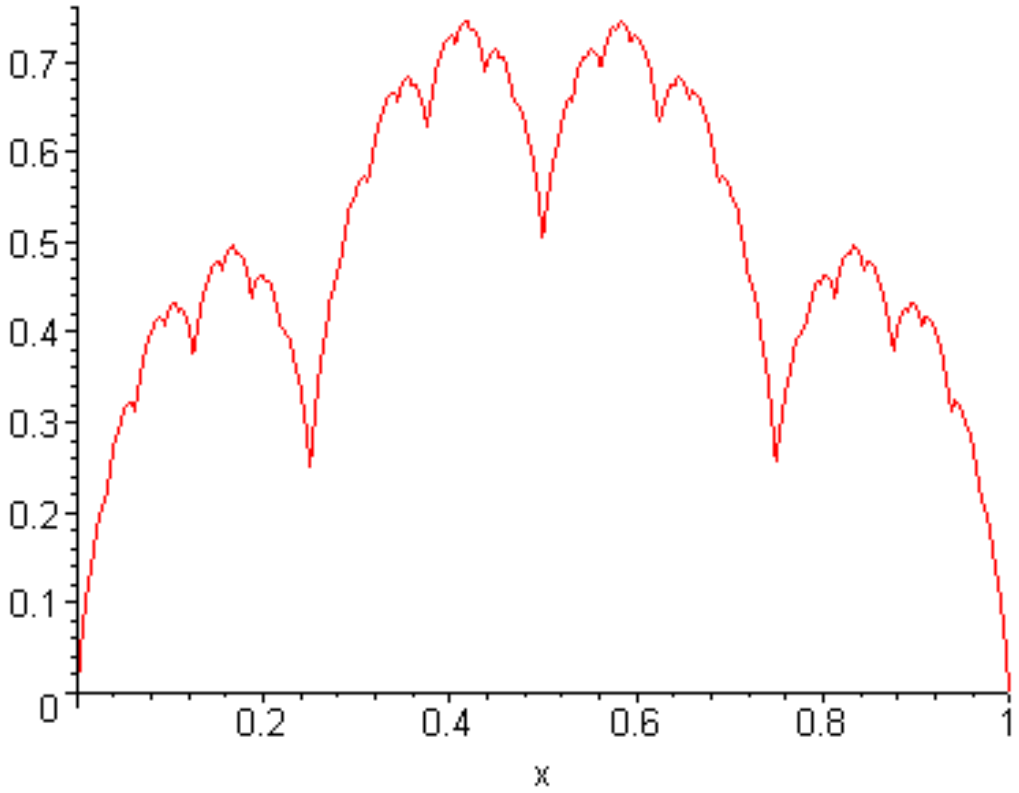

```
> plot('nowherediff(x,5)',x=0..1);
```



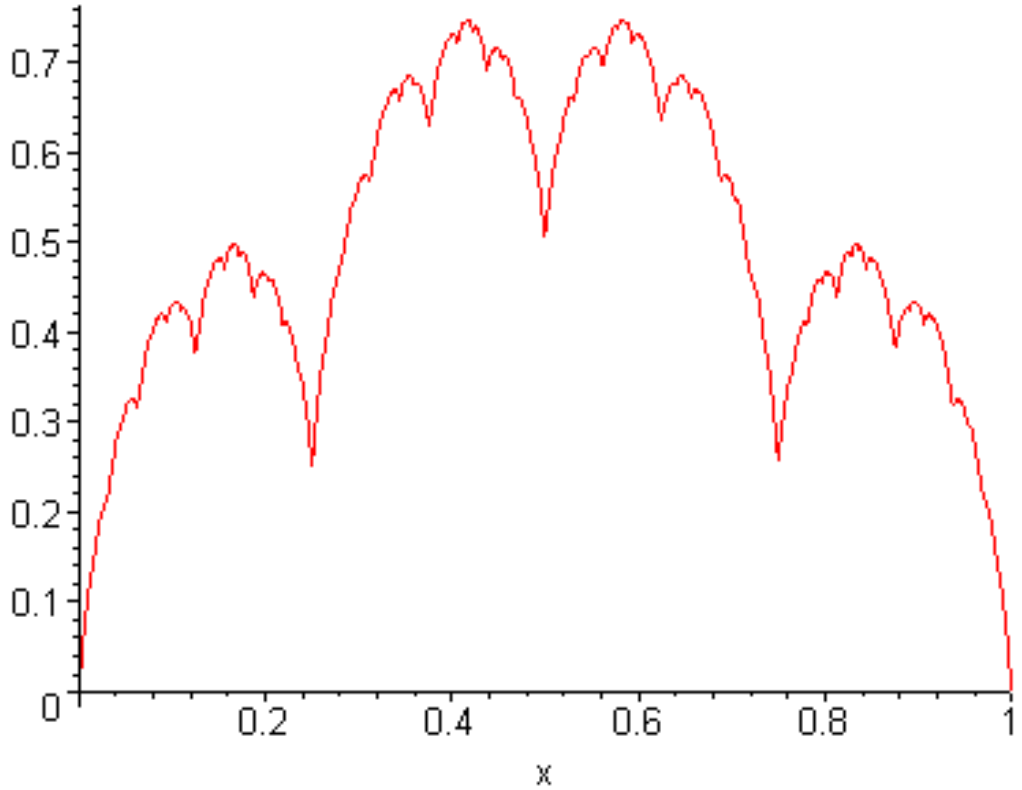

```
> plot('nowherediff(x,6)',x=0..1);
```




```
> plot('nowherediff(x,7)',x=0..1);
```




```
> plot('nowherediff(x,9)',x=0..1);
```



So why does the Van der Waerden Function fail to be differentiable? The main idea is that at each step of the sum, we add more peaks. The function will certainly not be differentiable at the top of each of these peaks. However, for any other point, what occurs is that we add so many peaks which occur on smaller and smaller intervals that we can get this point to be between two peaks, so just like the absolute value function at 0, the difference quotient at this point will have different limits depending on whether we take it from the left or the right.

We now change gears and move on to the topic of space filling curves. We know the interval of points [0,1] is an uncountable set. As such, it is impossible to find a function defined on the natural numbers that is onto the unit interval. However, it is a simple task to come up with a function that maps the unit interval onto the whole real line. We could take:

$f(x) := -\cot(\pi x)$

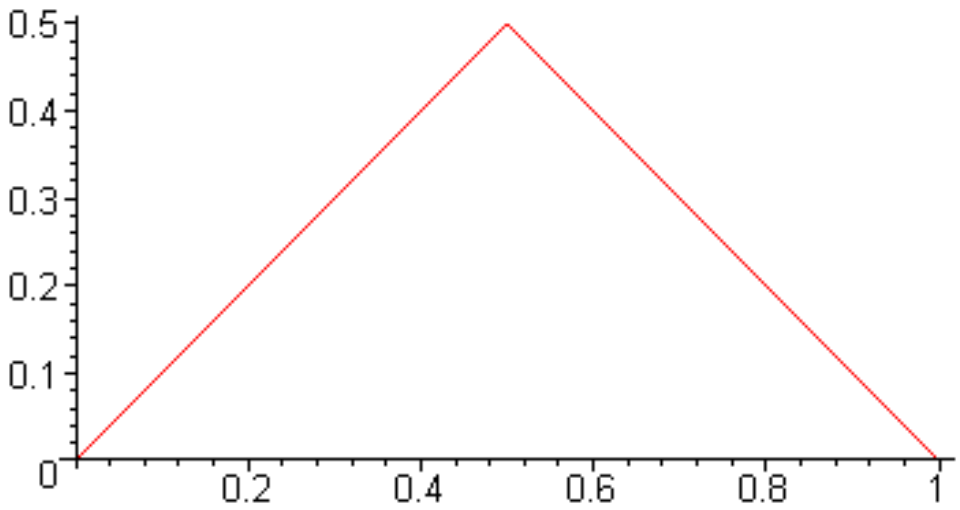
with f(0)=f(1)=1.

This shows that the cardinality of the unit segment is the same as the real number line. Now the question is, can we find a map from the unit segment [0,1] that is onto the unit square [0,1]x[0,1]? The answer is in the affirmative, as was shown by Giuseppe Peano and David Hilbert. The remarkable conclusion is that [0,1] and [0,1]x[0,1] are of the same size. Furthermore, through our discussion of the Cantor Function, it is trivial to see that Cantor Set can be mapped onto [0,1]. Thus, our misleadingly thin set the Cantor Set in reality has so many points in it that we can map the set onto [0,1]x[0,1]!

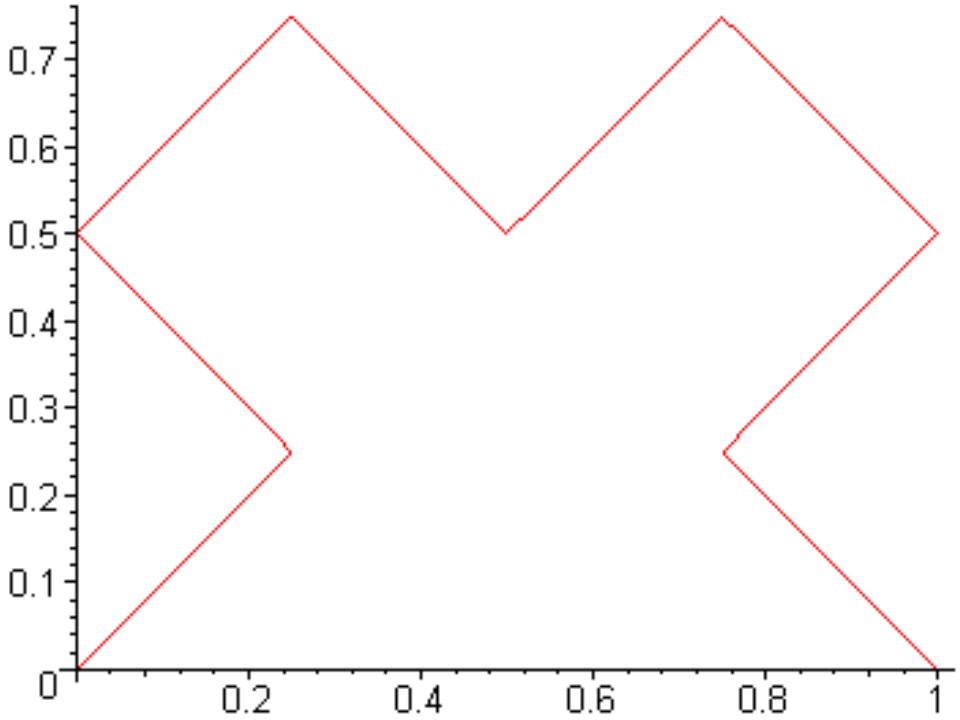
Such a curve that maps the unit interval onto the unit square is called a space-filling curve, and for good reason. However, Eugen Netto proved that any continuous map from [0,1] to [0,1]x[0,1] (or the unit cube) that is onto must fail to be injective. All of our examples will be onto and continuous, and we will readily see how they fail to be injective.

Peano's Space Filling Curve

Peano gave a wondeful example of a space-filling curve. Like the Cantor Function, the Peano Curve can be constructed as the limit of a sequence of curves. The first curve in the sequence maps [0,1] onto:

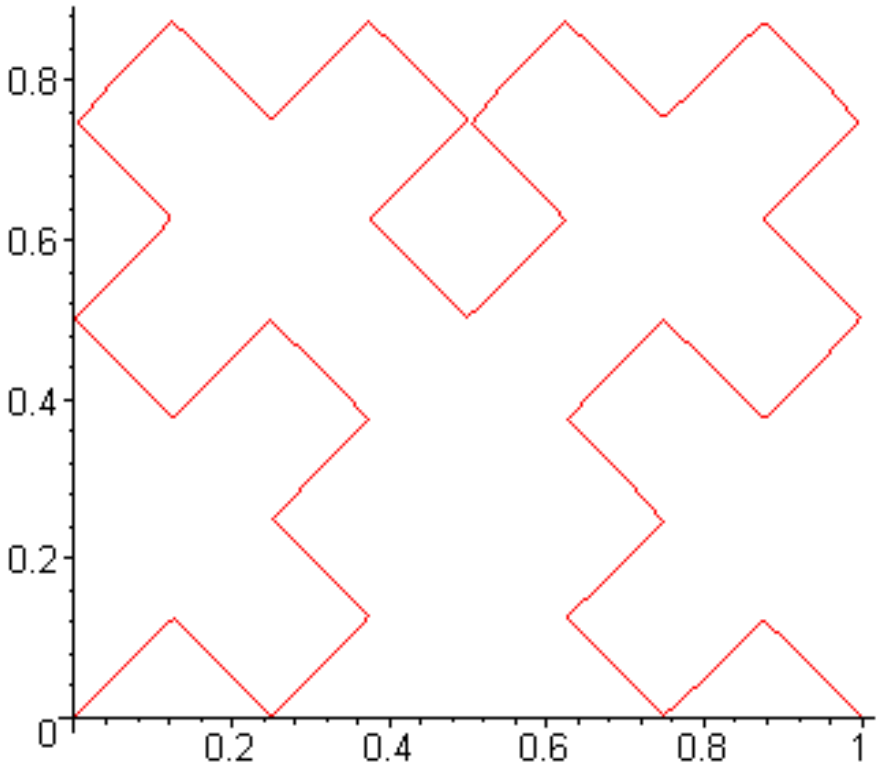


again, we reiterate that this is a vector-valued function, so that the above red line is the image of our function. The image of the second function in our sequence is:



What we have done is taken the unit square and divided it into four squares. We then took the image of our original function, halfed it, then we rotated it 90 degrees clockwise and drew it on the first square. This gives us the first one fourth of our above image. To get the second fourth, we again half the size of our first image, and this time translate the image up by a half. We get the last two fourths again by translations and rotations of the first image.

To get the image of the next function, we apply the same transformations to the second image. We will get:



One will note that the first fourth of this image is the previous one rotated clockwise by a right angle.

If we proceed as such, we will get a sequence of continuous functions which will converge uniformly to another continuous function. This limiting function, which we will call the Peano Curve, will turn out to be onto the unit square.

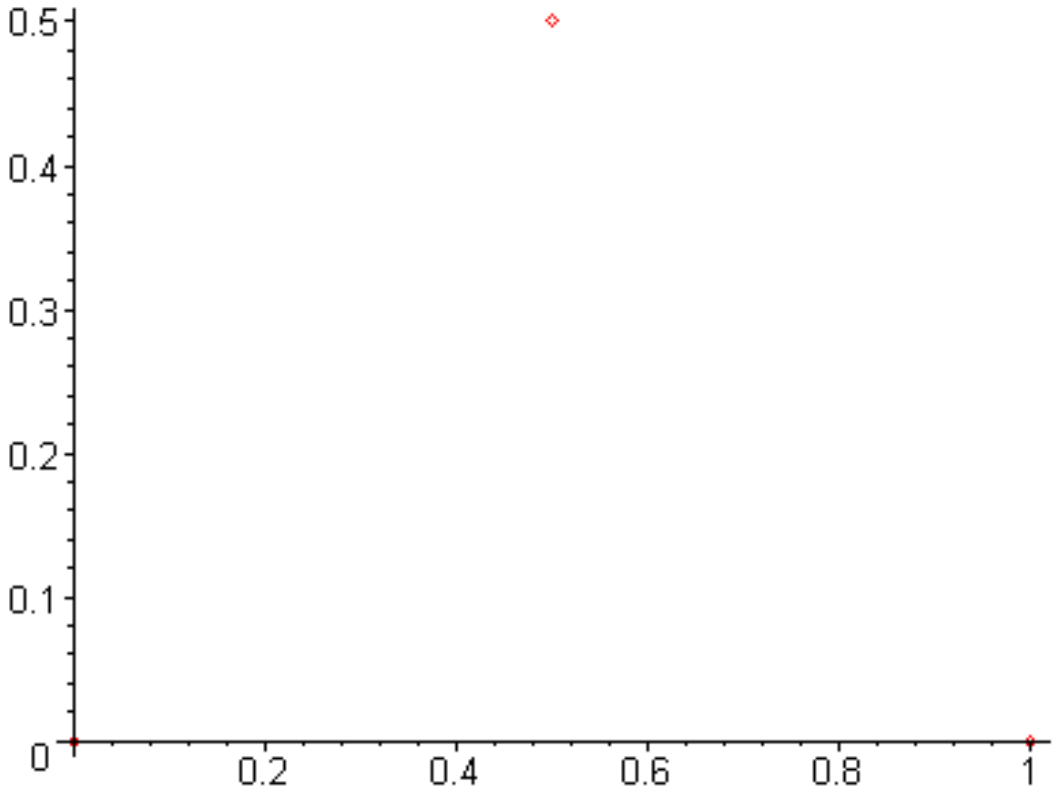
Our task now is to come up with an algorithm which will allow us to graph the sequence of functions which lead up to the Peano Curve. Our strategy is simple, we will first derive a procedure which lists for us the "peano points." We note from above that our sequence of functions consist of straight lines pieced together at a number of vertices. We call this sequence of vertices the peano points. If we can come up with a procedure that at **k** will list out the peano points at **k,** then producing the **k**th function in our sequence is merely connecting each of these vertices in the right order. We start by first manually producing the peano points for the first step:

```
> peanopoints1:=[[0,0],[1/2,1/2],[1,0]];
```

$$peanopoints1 := \left[[0, 0], \left[\frac{1}{2}, \frac{1}{2} \right], [1, 0] \right]$$

let us plot these points and see that indeed, if we join them in order we get the first function in our sequence:

```
> with(plots):
> pointplot(peanopoints1,color=red);
```



Now, to get the next set of peano points, we merely apply the four transformations to the previous points. We define these transformations now using:

```
> rotate:=(x,y)->[(x[1])*(cos(y))-
(x[2])*(sin(y)),(x[1])*(sin(y))+(x[2])*(cos(y))];
```

$$rotate := (x, y) \to [x_1 \cos(y) - x_2 \sin(y), x_1 \sin(y) + x_2 \cos(y)]$$

so that:

```
> peanotransform1:=(x)->(1/2)*rotate(x,-Pi/2)+[0,1/2];
```

$$peanotransform1 := x \to \frac{1}{2}\,\mathrm{rotate}\left(x, -\frac{1}{2}\pi\right) + \left[0, \frac{1}{2}\right]$$

```
> peanotransform2:=(x)->(1/2)*x+[0,1/2];
```

$$peanotransform2 := x \to \frac{1}{2}x + \left[0, \frac{1}{2}\right]$$

```
> peanotransform3:=(x)->(1/2)*x+[1/2,1/2];
```

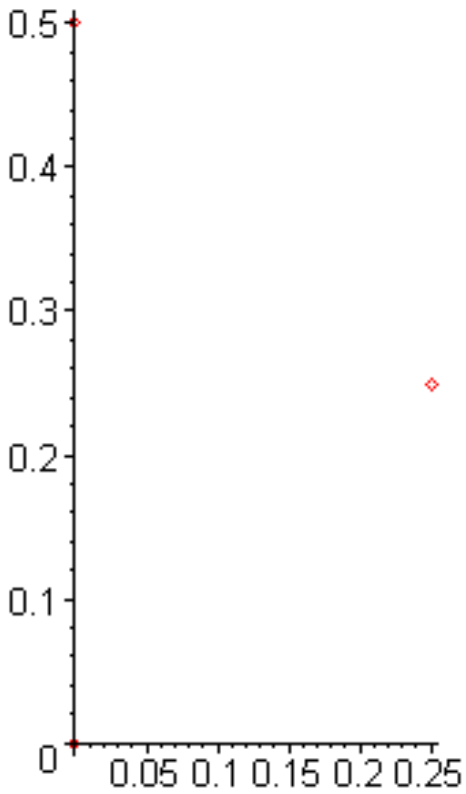
$$peanotransform3 := x \to \frac{1}{2}x + \left[\frac{1}{2}, \frac{1}{2}\right]$$

```
> peanotransform4:=(x)->(1/2)*rotate(x,Pi/2)+[1,0];
```

$$peanotransform4 := x \to \frac{1}{2}\,\mathrm{rotate}\left(x, \frac{1}{2}\pi\right) + [1, 0]$$

As a check, we apply the first transformation to the first peano points, which should give us the first three vertices of the second function in our sequence:

```
> pointplot(map(peanotransform1,peanopoints1),color=red);
```



Now, let me display my algorithm:

```
> peanopoint:=proc(i,k)
option remember;
if k=1 then (peanopoints1)[i+1];
else if i=0 then [0,0];
else if i <= (2^(2*k-3)) then
map(peanotransform1,[seq(peanopoint(j,k-1),j=0..(2^(2*k-
3)))])[(2^(2*k-3))-i+1];
else if i <= (2^(2*k-2)) then
map(peanotransform2,[seq(peanopoint(j,k-1),j=0..(2^(2*k-
3)))])[i-(2^(2*k-3))+1];
else if i <= 3*(2^(2*k-3)) then
map(peanotransform3,[seq(peanopoint(j,k-1),j=0..(2^(2*k-
3)))])[i-(2^(2*k-2))+1];
else
map(peanotransform4,[seq(peanopoint(j,k-1),j=0..(2^(2*k-
3)))])[(2^(2*k-1))-i+1];
fi; fi; fi; fi; fi; end;
```

```
peanopoint := proc(i, k)
option remember;
    if k = 1 then peanopoints1[i + 1]
    else
        if i = 0 then [0, 0]
```

```
        else
            if i ≤ 2^(2×k − 3) then map(peanotransform1,
                [seq(peanopoint(j, k − 1), j = 0 .. 2^(2×k − 3))])[
                2^(2×k − 3) − i + 1]
            else
                if i ≤ 2^(2×k − 2) then map(peanotransform2,
                    [seq(peanopoint(j, k − 1), j = 0 .. 2^(2×k − 3))])[
                    i − 2^(2×k − 3) + 1]
                else
                    if i ≤ 3×2^(2×k − 3) then map(peanotransform3,
                        [seq(peanopoint(j, k − 1), j = 0 .. 2^(2×k − 3))])[
                        i − 2^(2×k − 2) + 1]
                    else map(peanotransform4,
                        [seq(peanopoint(j, k − 1), j = 0 .. 2^(2×k − 3))])[
                        2^(2×k − 1) − i + 1]
                    end if end proc
                end if
            end if
        end if
    end if
```

Again, this algorithm is recursive and is based on the four transformations. Given an **i** and a **k**, this algorithm gives the **i**th peano point of the **k**th function in our sequence, the order in which the algorithm gives our points being in the order that we later will want to connect them. Here is how it works, one can check that at each step **k**, there are 2^(2**k**-1)+1 peano points. What we do is divide the task of finding the current peano points from the previous set into five jobs. To get the peano points at the **k**th step, we start by setting the first point (at **i**=0) to (0,0). Then, we get the first fourth of the remaining points by taking the points at the step **k**-1, applying the first transformation, and then taking those points in reverse order. We then get the second, third and last fourth of the points by applying the second, third and fourth transformations respectively to the points from step **k**-1, and also taking them in reverse order.

At this point, we can again write a procedure which lists out all of the points at a step:

```
> peanopointset:=(k)->[seq(peanopoint(i,k),i=0..(2^(2*k-1)))];
```

$$peanopointset := k \rightarrow [\mathrm{seq}(\mathrm{peanopoint}(i, k), i = 0 .. 2^{(2k-1)})]$$

and now we list the points for several **k**:

```
> peanopointset(1);
```

$$\left[[0, 0], \left[\frac{1}{2}, \frac{1}{2}\right], [1, 0]\right]$$

```
> peanopointset(2);
```

$$\left[[0,0],\left[\frac{1}{4},\frac{1}{4}\right],\left[0,\frac{1}{2}\right],\left[\frac{1}{4},\frac{3}{4}\right],\left[\frac{1}{2},\frac{1}{2}\right],\left[\frac{3}{4},\frac{3}{4}\right],\left[1,\frac{1}{2}\right],\left[\frac{3}{4},\frac{1}{4}\right],[1,0]\right]$$

> **peanopointset(3);**

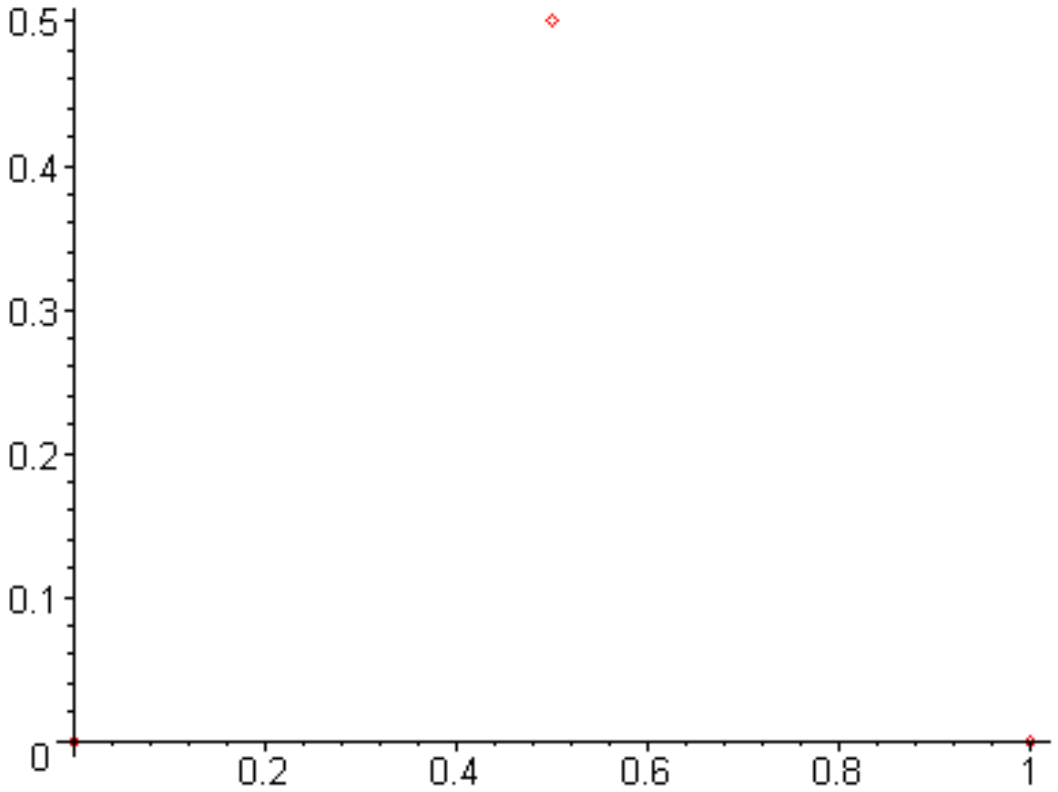
$$\left[[0,0],\left[\frac{1}{8},\frac{1}{8}\right],\left[\frac{1}{4},0\right],\left[\frac{3}{8},\frac{1}{8}\right],\left[\frac{1}{4},\frac{1}{4}\right],\left[\frac{3}{8},\frac{3}{8}\right],\left[\frac{1}{4},\frac{1}{2}\right],\left[\frac{1}{8},\frac{3}{8}\right],\left[0,\frac{1}{2}\right],\left[\frac{1}{8},\frac{5}{8}\right],\left[0,\frac{3}{4}\right],\left[\frac{1}{8},\frac{7}{8}\right],\left[\frac{1}{4},\frac{3}{4}\right],\left[\frac{3}{8},\frac{7}{8}\right],\left[\frac{1}{2},\frac{3}{4}\right],\left[\frac{3}{8},\frac{5}{8}\right],\left[\frac{1}{2},\frac{1}{2}\right],\left[\frac{5}{8},\frac{5}{8}\right],\left[\frac{1}{2},\frac{3}{4}\right],\left[\frac{5}{8},\frac{7}{8}\right],\left[\frac{3}{4},\frac{3}{4}\right],\left[\frac{7}{8},\frac{7}{8}\right],\left[1,\frac{3}{4}\right],\left[\frac{7}{8},\frac{5}{8}\right],\left[1,\frac{1}{2}\right],\left[\frac{7}{8},\frac{3}{8}\right],\left[\frac{3}{4},\frac{1}{2}\right],\left[\frac{5}{8},\frac{3}{8}\right],\left[\frac{3}{4},\frac{1}{4}\right],\left[\frac{5}{8},\frac{1}{8}\right],\left[\frac{3}{4},0\right],\left[\frac{7}{8},\frac{1}{8}\right],[1,0]\right]$$

> **peanopointset(4);**

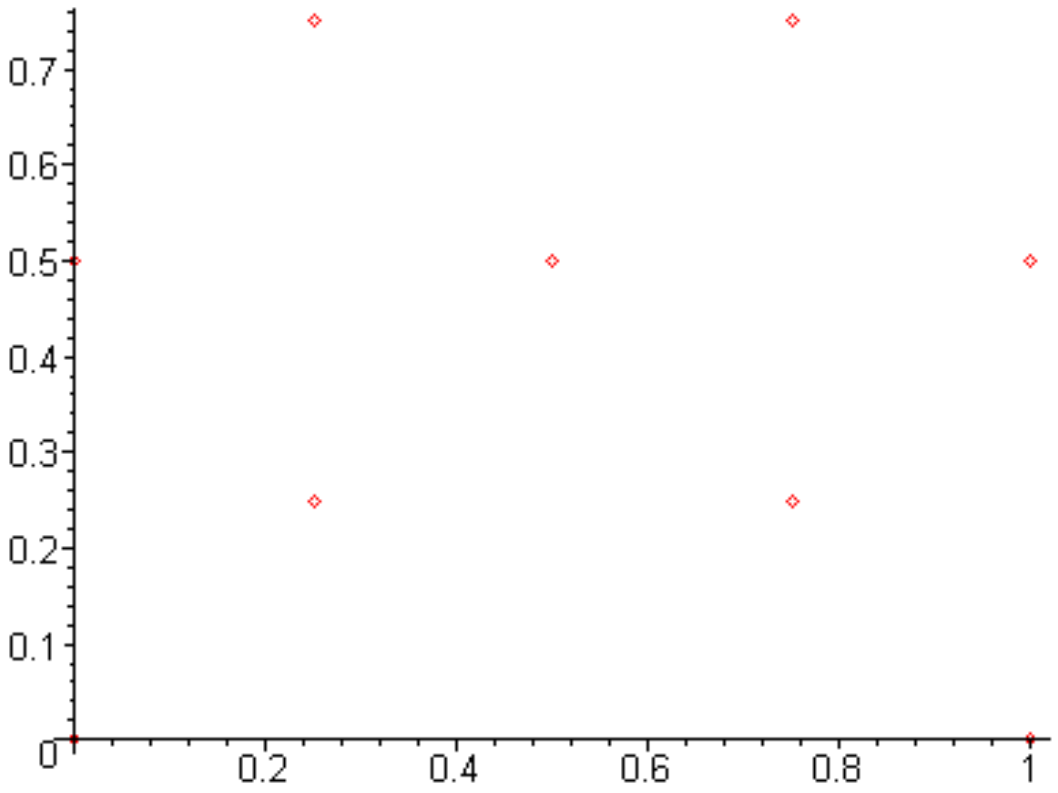
$$\left[[0,0],\left[\frac{1}{16},\frac{1}{16}\right],\left[0,\frac{1}{8}\right],\left[\frac{1}{16},\frac{3}{16}\right],\left[\frac{1}{8},\frac{1}{8}\right],\left[\frac{3}{16},\frac{3}{16}\right],\left[\frac{1}{4},\frac{1}{8}\right],\left[\frac{3}{16},\frac{1}{16}\right],\left[\frac{1}{4},0\right],\left[\frac{5}{16},\frac{1}{16}\right],\left[\frac{3}{8},0\right],\left[\frac{7}{16},\frac{1}{16}\right],\left[\frac{3}{8},\frac{1}{8}\right],\left[\frac{7}{16},\frac{3}{16}\right],\left[\frac{3}{8},\frac{1}{4}\right],\left[\frac{5}{16},\frac{3}{16}\right],\left[\frac{1}{4},\frac{1}{4}\right],\left[\frac{5}{16},\frac{5}{16}\right],\left[\frac{3}{8},\frac{1}{4}\right],\left[\frac{7}{16},\frac{5}{16}\right],\left[\frac{3}{8},\frac{3}{8}\right],\left[\frac{7}{16},\frac{7}{16}\right],\left[\frac{3}{8},\frac{1}{2}\right],\left[\frac{5}{16},\frac{7}{16}\right],\left[\frac{1}{4},\frac{1}{2}\right],\left[\frac{3}{16},\frac{7}{16}\right],\left[\frac{1}{4},\frac{3}{8}\right],\left[\frac{3}{16},\frac{5}{16}\right],\left[\frac{1}{8},\frac{3}{8}\right],\left[\frac{1}{16},\frac{5}{16}\right],\left[0,\frac{3}{8}\right],\left[\frac{1}{16},\frac{7}{16}\right],\left[0,\frac{1}{2}\right],\left[\frac{1}{16},\frac{9}{16}\right],\left[\frac{1}{8},\frac{1}{2}\right],\left[\frac{3}{16},\frac{9}{16}\right],\left[\frac{1}{8},\frac{5}{8}\right],\left[\frac{3}{16},\frac{11}{16}\right],\left[\frac{1}{8},\frac{3}{4}\right],\left[\frac{1}{16},\frac{11}{16}\right],\left[0,\frac{3}{4}\right],\left[\frac{1}{16},\frac{13}{16}\right],\left[0,\frac{7}{8}\right],\left[\frac{1}{16},\frac{15}{16}\right],\left[\frac{1}{8},\frac{7}{8}\right],\left[\frac{3}{16},\frac{15}{16}\right],\left[\frac{1}{4},\frac{7}{8}\right],\left[\frac{3}{16},\frac{13}{16}\right],\left[\frac{1}{4},\frac{3}{4}\right],\left[\frac{5}{16},\frac{13}{16}\right],\left[\frac{1}{4},\frac{7}{8}\right],\left[\frac{5}{16},\frac{15}{16}\right],\left[\frac{3}{8},\frac{7}{8}\right],\left[\frac{7}{16},\frac{15}{16}\right],\left[\frac{1}{2},\frac{7}{8}\right],\left[\frac{7}{16},\frac{13}{16}\right],\left[\frac{1}{2},\frac{3}{4}\right],\left[\frac{7}{16},\frac{11}{16}\right],\left[\frac{3}{8},\frac{3}{4}\right],\left[\frac{5}{16},\frac{11}{16}\right],\left[\frac{3}{8},\frac{5}{8}\right],\left[\frac{5}{16},\frac{9}{16}\right],\left[\frac{3}{8},\frac{1}{2}\right],\left[\frac{7}{16},\frac{9}{16}\right],\left[\frac{1}{2},\frac{1}{2}\right],\left[\frac{9}{16},\frac{9}{16}\right],\left[\frac{5}{8},\frac{1}{2}\right],\left[\frac{11}{16},\frac{9}{16}\right],\left[\frac{5}{8},\frac{5}{8}\right],\left[\frac{11}{16},\frac{11}{16}\right],\left[\frac{5}{8},\frac{3}{4}\right],\left[\frac{9}{16},\frac{11}{16}\right],\left[\frac{1}{2},\frac{3}{4}\right],\left[\frac{9}{16},\frac{13}{16}\right],\left[\frac{1}{2},\frac{7}{8}\right],\left[\frac{9}{16},\frac{15}{16}\right],\left[\frac{5}{8},\frac{7}{8}\right],\left[\frac{11}{16},\frac{15}{16}\right],\left[\frac{3}{4},\frac{7}{8}\right],\left[\frac{11}{16},\frac{13}{16}\right],\left[\frac{3}{4},\frac{3}{4}\right],\left[\frac{13}{16},\frac{13}{16}\right],\left[\frac{3}{4},\frac{7}{8}\right],\left[\frac{13}{16},\frac{15}{16}\right],\left[\frac{7}{8},\frac{7}{8}\right],\left[\frac{15}{16},\frac{15}{16}\right],\left[1,\frac{7}{8}\right],\left[\frac{15}{16},\frac{13}{16}\right],\left[1,\frac{3}{4}\right],\left[\frac{15}{16},\frac{11}{16}\right],\left[\frac{7}{8},\frac{3}{4}\right],\left[\frac{13}{16},\frac{11}{16}\right],\left[\frac{7}{8},\frac{5}{8}\right],\left[\frac{13}{16},\frac{9}{16}\right],\left[\frac{7}{8},\frac{1}{2}\right],\left[\frac{15}{16},\frac{9}{16}\right],\left[1,\frac{1}{2}\right],\left[\frac{15}{16},\frac{7}{16}\right],\left[1,\frac{3}{8}\right],\left[\frac{15}{16},\frac{5}{16}\right],\left[\frac{7}{8},\frac{3}{8}\right],\left[\frac{13}{16},\frac{5}{16}\right],\left[\frac{3}{4},\frac{3}{8}\right],\left[\frac{13}{16},\frac{7}{16}\right],\left[\frac{3}{4},\frac{1}{2}\right],\left[\frac{11}{16},\frac{7}{16}\right],\left[\frac{5}{8},\frac{1}{2}\right],\left[\frac{9}{16},\frac{7}{16}\right],\left[\frac{5}{8},\frac{3}{8}\right],\left[\frac{9}{16},\frac{5}{16}\right],\left[\frac{5}{8},\frac{1}{4}\right],\left[\frac{11}{16},\frac{5}{16}\right],\left[\frac{3}{4},\frac{1}{4}\right],\left[\frac{11}{16},\frac{3}{16}\right],\left[\frac{5}{8},\frac{1}{4}\right],\left[\frac{9}{16},\frac{3}{16}\right],\left[\frac{5}{8},\frac{1}{8}\right],\left[\frac{9}{16},\frac{1}{16}\right],\left[\frac{5}{8},0\right],\left[\frac{11}{16},\frac{1}{16}\right],\left[\frac{3}{4},0\right],\left[\frac{13}{16},\frac{1}{16}\right],\left[\frac{3}{4},\frac{1}{8}\right],\left[\frac{13}{16},\frac{3}{16}\right],\left[\frac{7}{8},\frac{1}{8}\right],\left[\frac{15}{16},\frac{3}{16}\right],\left[1,\frac{1}{8}\right],\left[\frac{15}{16},\frac{1}{16}\right],[1,0]\right]$$

and we also plot them:

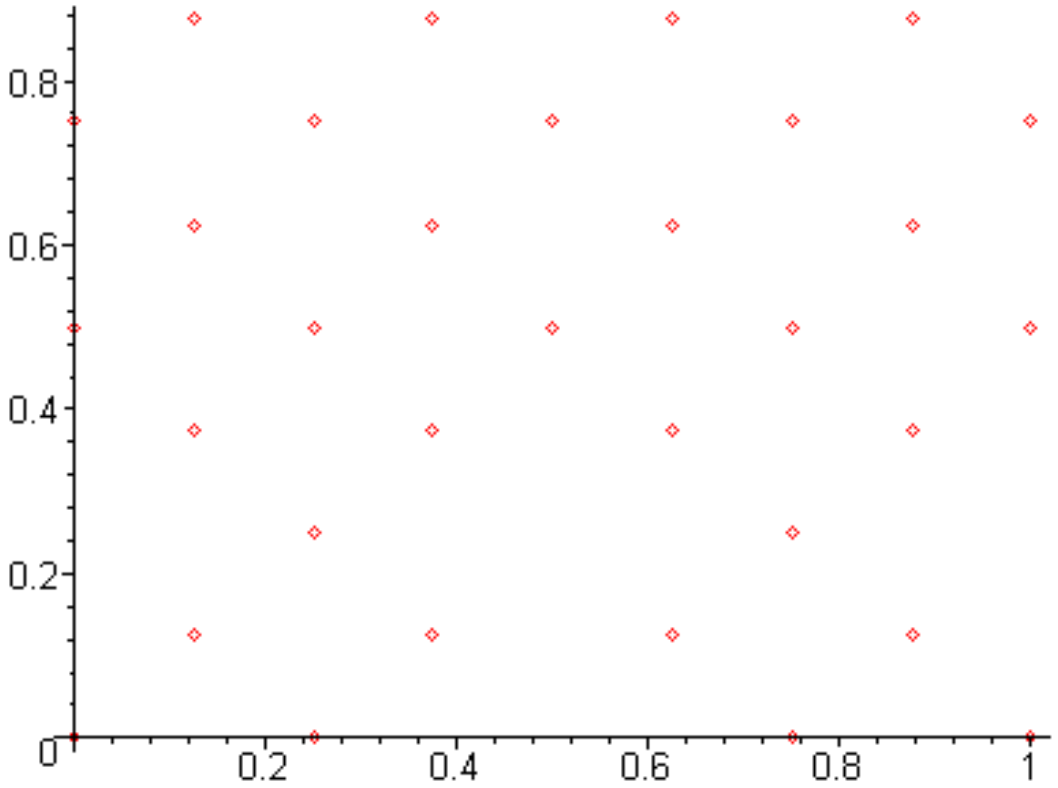
```
> pointplot(peanopointset(1),color=red);
```



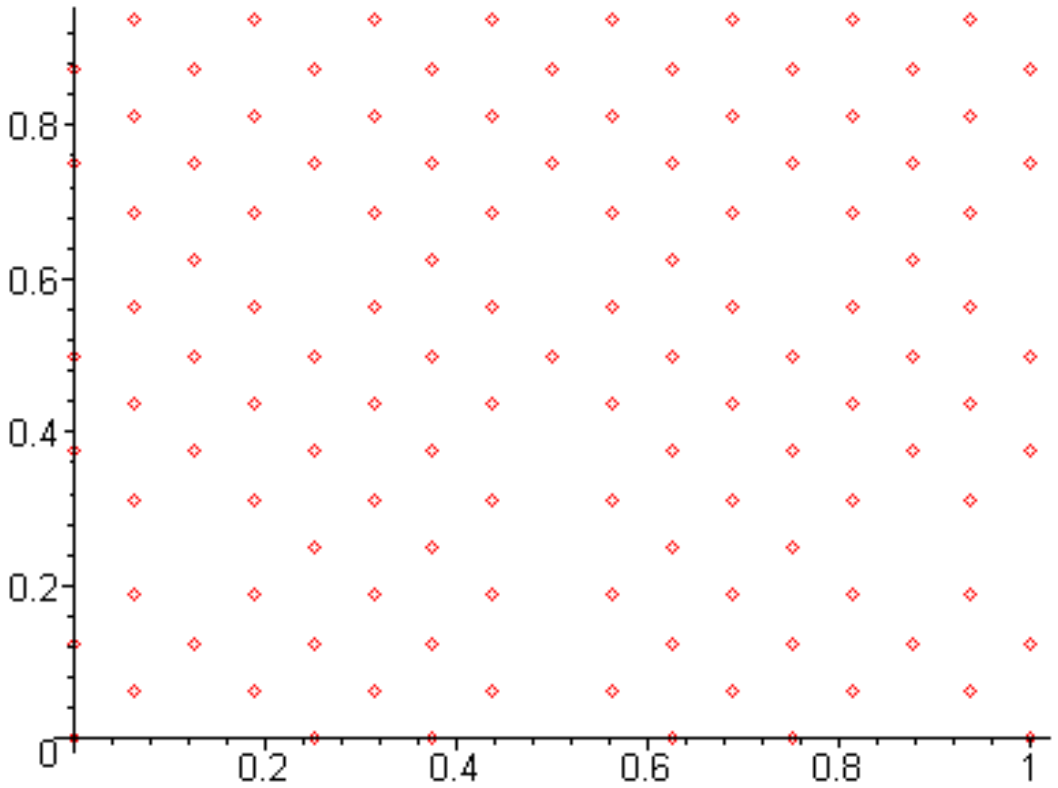

```
> pointplot(peanopointset(2),color=red);
```



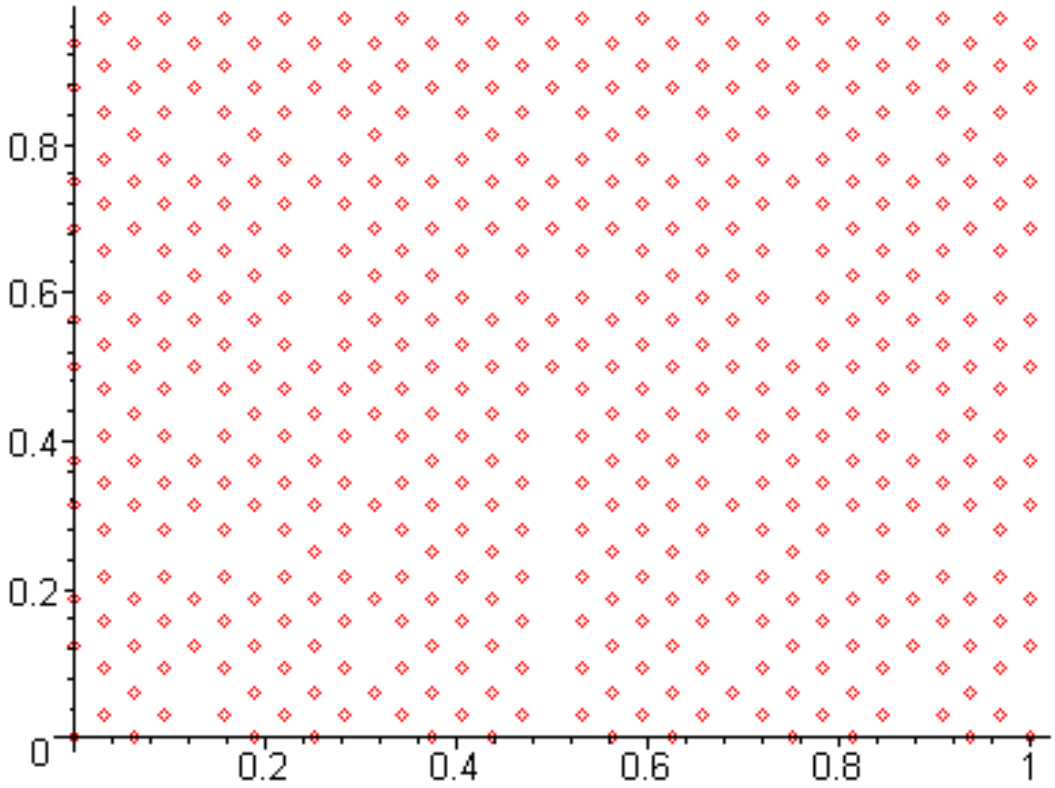

```
> pointplot(peanopointset(3),color=red);
```




```
> pointplot(peanopointset(4),color=red);
```




```
> pointplot(peanopointset(5),color=red);
```



What one expects from these plots is to be convinced that the peano points at the kth step include all of the peano points for all of the previous steps. Furthermore, the union over all **k** of the peano points at **k** is a dense set in the unit square. Lastly and more importantly, each of our curves in our sequence goes through the peano points for all of the steps before it, so that our limiting function must take as its values the union over **k** of all of the peano points, which is a dense set on [0,1]x[0,1]. Now, the Peano Curve will be continuous, and so must take the compact set [0,1] to a compact and hence closed subset of [0,1]x[0,1]. However, a dense set is part of its image, which is the set of all peano points. This loosely shows that our Peano Curve is onto the unit square.

Given our peano point algorithm, it is now an easy business to construct our sequence of functions. Since our sequence of functions are merely straightlines pieced together, we repeat our trick of first defining each piece seperately for each step. This results in:

```
> peanocurvepart:=(t,i,k)->(peanopoint(i+1,k)-
peanopoint(i,k))*t+peanopoint(i,k);
```

peanocurvepart :=

$$(t, i, k) \rightarrow (\text{peanopoint}(i + 1, k) - \text{peanopoint}(i, k))\, t + \text{peanopoint}(i, k)$$

With each piece defined as above, we will not be able to simply sum each piece to get our **k**th function. Thus, we will now illustrate the second solution in defining a sequence of piecewise defined functions, which involves using a do loop. Here is our algorithm:

```
> peanocurve:=proc(t,k)
if t=1
then [1,0] else
for i from 0 to (2^(2*k-1)) do
```

```
if (i/(2^(2*k-1))) <= t and t < ((i+1)/(2^(2*k-1))) then
peanocurvepart((t*(2^(2*k-1)))-i,i,k); break; else fi; od;
fi; end;
Warning, `i` is implicitly declared local to procedure `peanocurve`
```

peanocurve := **proc** (*t*, *k*)
local *i*;
if *t* = 1 **then** [1, 0]
else for *i* **from** 0 **to** 2^(2×*k* − 1) **do**
if *i*/2^(2×*k* − 1) ≤ *t* **and** *t* < (*i* + 1)/2^(2×*k* − 1) **then**
peanocurvepart(*t*×2^(2×*k* − 1) − *i*, *i*, *k*); **break** **end proc**
else
end if
end do
end if

This algorithm works as follows, we are given a **t** and a **k**, and we want to evaulate the **k**th function in our sequence at **t**. Now, the **k**th function in our sequence is composed of 2^(2**k**-1) straight lines which are consecutive peano points joined together. What this algorithm does is divide the unit interval [0,1] into 2^(2**k**-1) pieces. Then in a do loop, this algorithm determines in which of these pieces **t** lies in. Once it has determined that t lies in the ith interval, it assigns to it the value corresponding to the line connecting the **i**th and **i**+1th peano point. Let us see the algorithm at work and graph some curves. First we must define:

> `peanocurve1:=(t,k)->peanocurve(t,k)[1];`

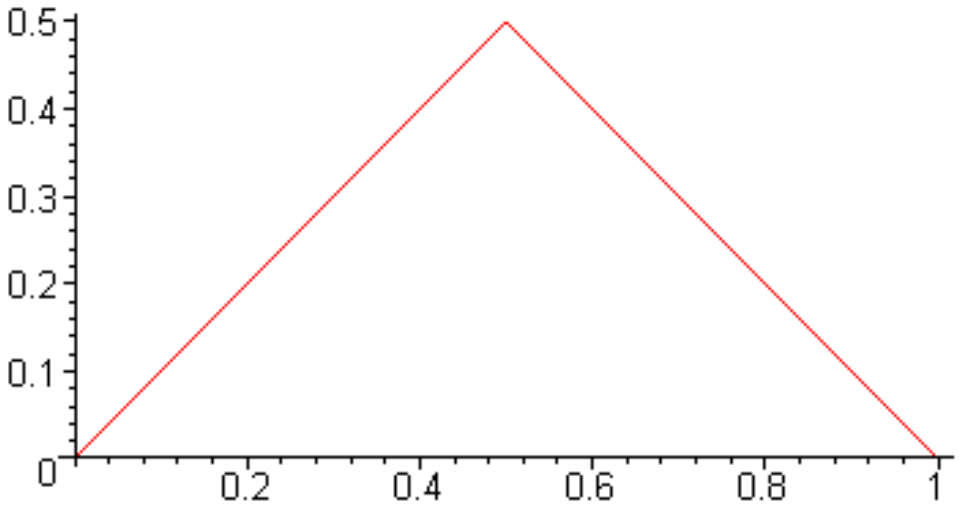
peanocurve1 := (*t*, *k*) → peanocurve(*t*, *k*)$_1$

> `peanocurve2:=(t,k)->peanocurve(t,k)[2];`

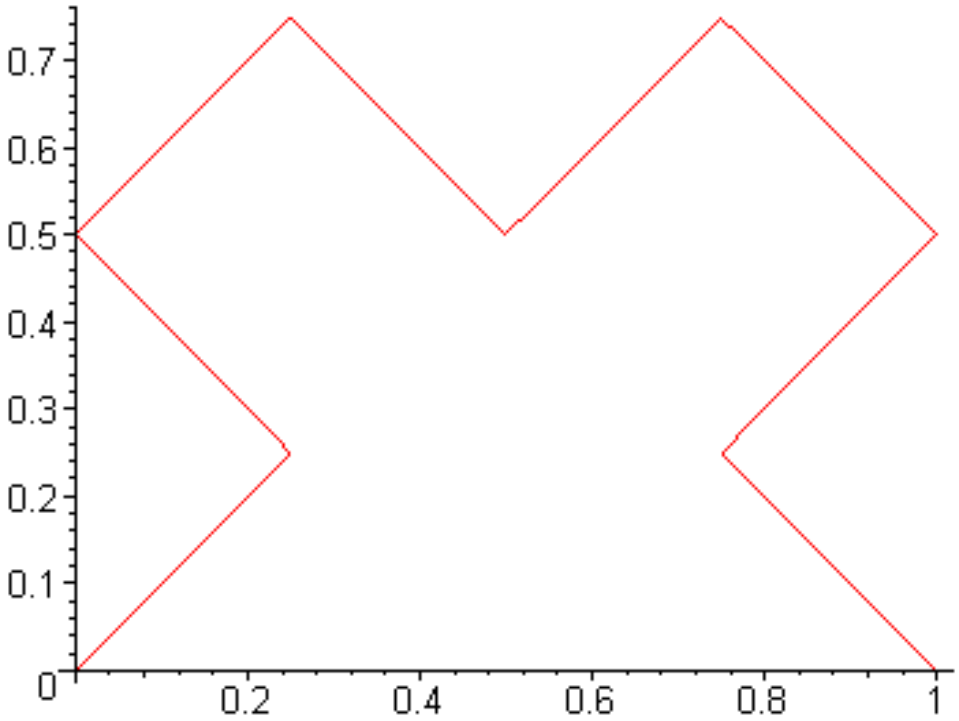
peanocurve2 := (*t*, *k*) → peanocurve(*t*, *k*)$_2$

So that we can parametrically graph our curves. Let us see the curves for **k** up to six:

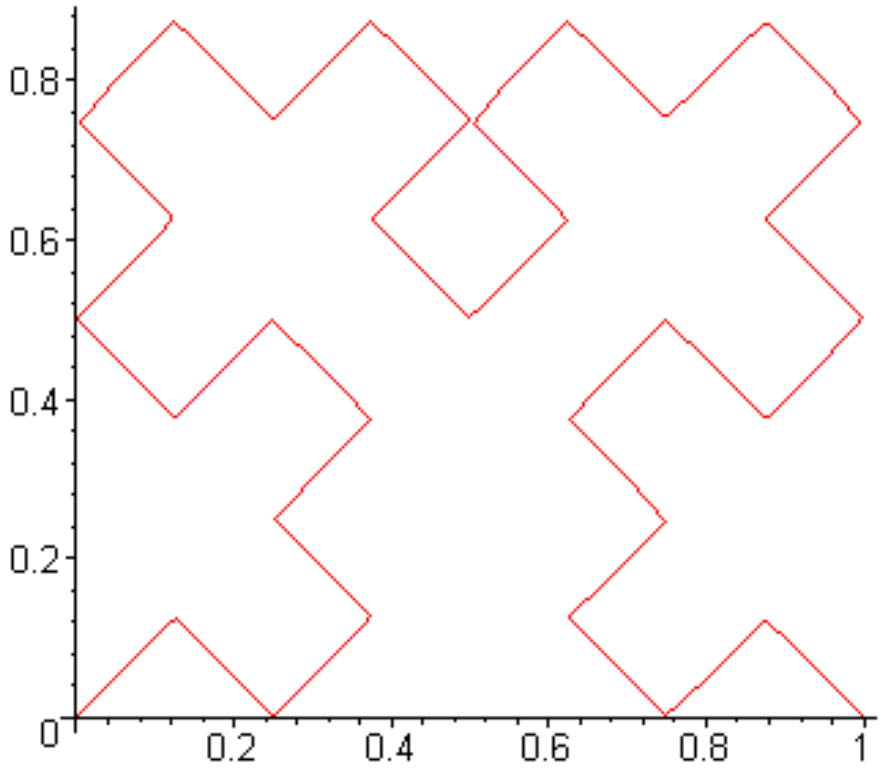
> `plot(['peanocurve1(t,1)','peanocurve2(t,1)',t=0..1]);`




```
> plot(['peanocurve1(t,2)','peanocurve2(t,2)',t=0..1]);
```

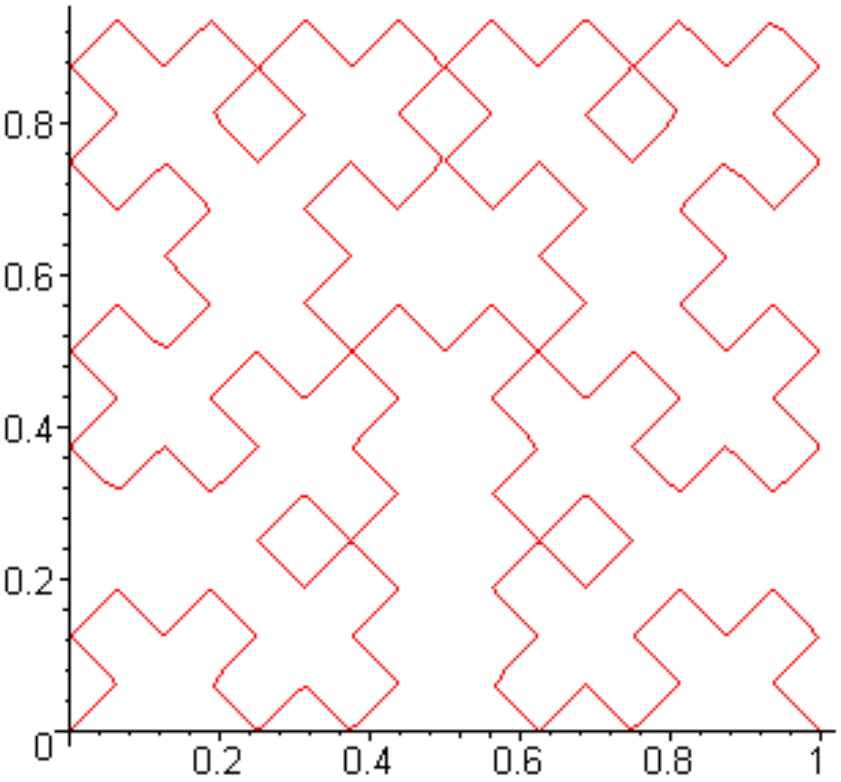



```
> plot(['peanocurve1(t,3)','peanocurve2(t,3)',t=0..1]);
```

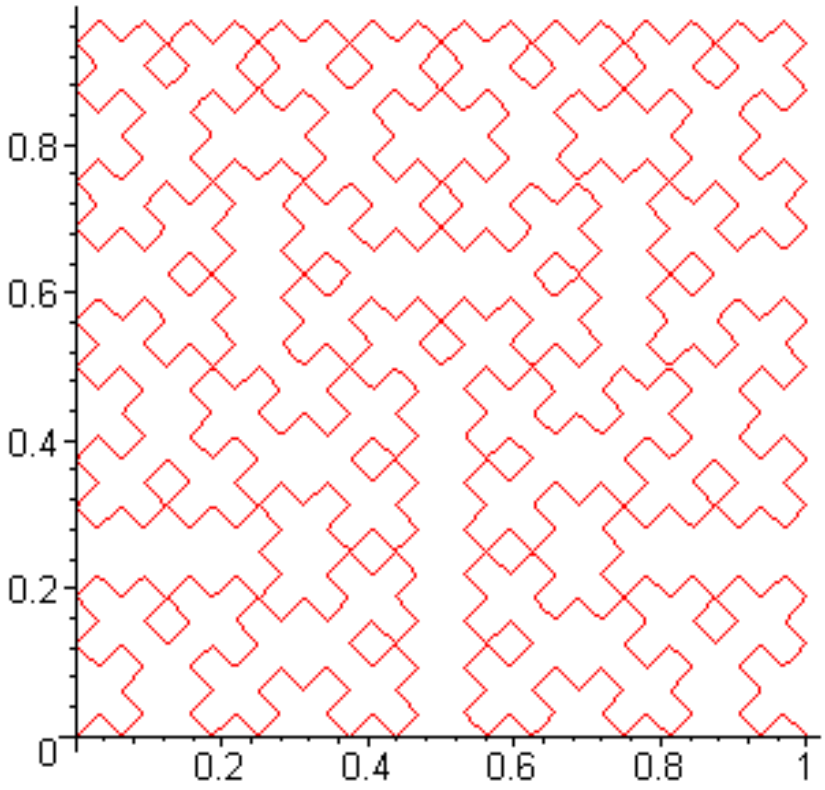


>

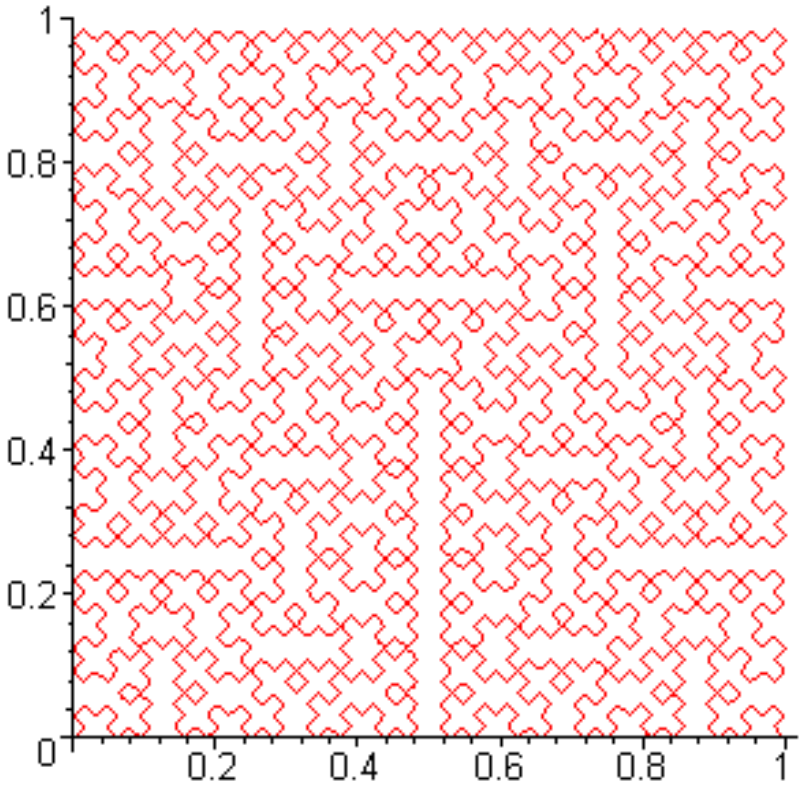
```
plot(['peanocurve1(t,4)','peanocurve2(t,4)',t=0..1],numpoin
ts=200);
```



```
>
plot(['peanocurve1(t,5)','peanocurve2(t,5)',t=0..1],numpoin
ts=513);
```




```
>
plot(['peanocurve1(t,6)','peanocurve2(t,6)',t=0..1],numpoin
ts=2049);
```



We note that after **k**=3, our functions are no longer injective, and where one curve in our sequence failed to be injective, the curves after it also failed. Thus, our limiting function will not be injective. We also note the increasing amount of iterations needed to produce a respectable graph.

Hopefully, one is convinced from the graphs that the Peano Curve will be onto and continuous. However, is it differentiable? The answer is in the extreme negative. The Peano Curve as it turns out, is nowhere differentiable.
Our next example of a space-filling curve is due to Hilbert. It can also be constructed as the limit of a sequence of function. Like the Peano Curve, the Hilbert Curve will be continuous and nowhere differentiable, and onto but not injective. Let us explore the algorithm derived to produce the Hilbert Curve.

Hilbert's Space Filling Curve

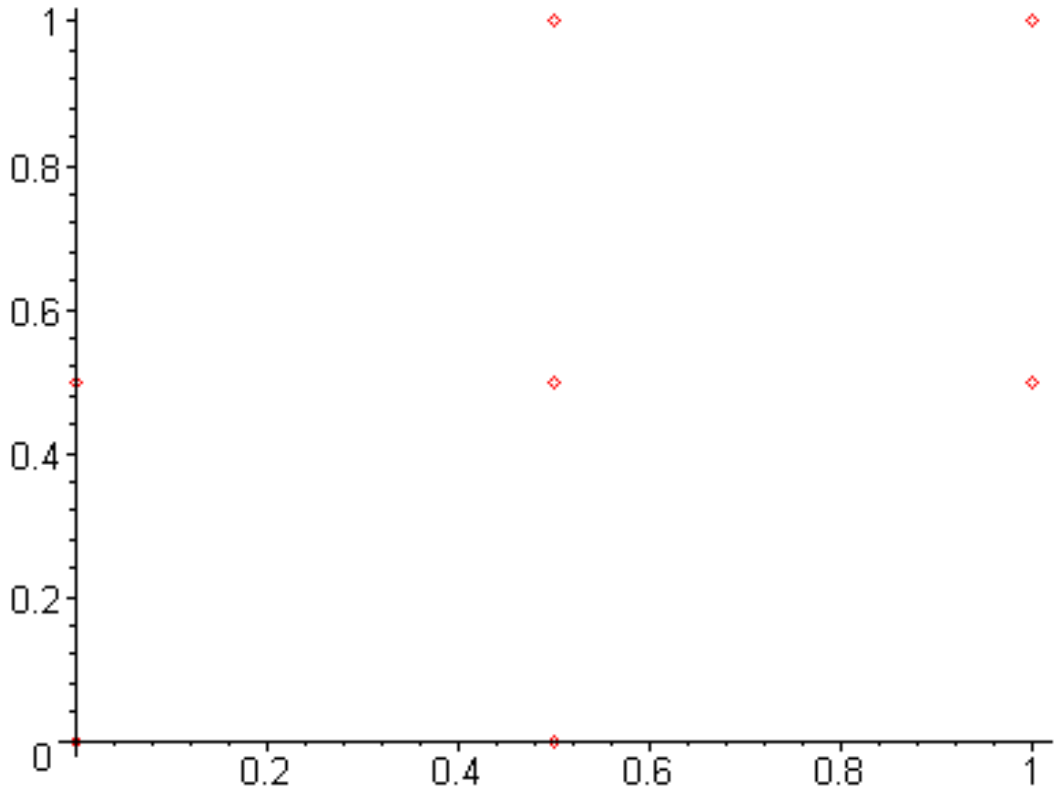
As in the Peano Curve, we first start with a small set of points:

```
>
hilbertpoints1:=[[0,0],[1/2,1/2],[0,1/2],[1/2,1],[1/2,0],[1
,1/2],[1/2,1/2],[1,1]];
```

$$hilbertpoints1 := \left[[0,0],\left[\frac{1}{2},\frac{1}{2}\right],\left[0,\frac{1}{2}\right],\left[\frac{1}{2},1\right],\left[\frac{1}{2},0\right],\left[1,\frac{1}{2}\right],\left[\frac{1}{2},\frac{1}{2}\right],[1,1]\right]$$

Now, if we connect these points in order we will get the first function in our sequence of functions leading up to the Hilbert Curve. Let us plot the points:

```
> with(plots):
> pointplot(hilbertpoints1,color=red);
```



Now, let us define the following three transformations:

```
> hilberttransform1:=(x)->(1/2)*x+[0,1/2];
```

$$hilberttransform1 := x \to \frac{1}{2}x + \left[0, \frac{1}{2}\right]$$

```
> hilberttransform2:=(x)->(1/2)*x+[1/2,0];
```

$$hilberttransform2 := x \to \frac{1}{2}x + \left[\frac{1}{2}, 0\right]$$

```
> hilberttransform3:=(x)->(1/2)*x+[1/2,1/2];
```

$$hilberttransform3 := x \to \frac{1}{2}x + \left[\frac{1}{2}, \frac{1}{2}\right]$$

Using these transformations, define the following algorithm:

```
> hilbertpoint:=proc(i,k)
option remember;
if k=1 then (hilbertpoints1)[i]
else
if i <= 2^(2*k-1)
then ((1/2)*[seq(hilbertpoint(j,k-1),j=1..(2^(2*k-1)))])[i]
;
else
if i <= 2^(2*k)
then map(hilberttransform1,[seq(hilbertpoint(j,k-
1),j=1..(2^(2*k-1)))])[i-(2^(2*k-1))] ;
else
```

```
if i <= 3*(2^(2*k-1))
then map(hilberttransform2,[seq(hilbertpoint(j,k-
1),j=1..(2^(2*k-1)))])[i-(2^(2*k))] ;
else
map(hilberttransform3,[seq(hilbertpoint(j,k-
1),j=1..(2^(2*k-1)))])[i-(3*(2^(2*k-1)))] ;
fi; fi; fi; fi; end;
```

```
hilbertpoint := proc(i, k)
option remember;
    if k = 1 then hilbertpoints1[i]
    else
        if i ≤ 2^(2×k − 1) then
            [1/2×seq(hilbertpoint(j, k − 1), j = 1 .. 2^(2×k − 1))][i]
        else
            if i ≤ 2^(2×k) then map(hilberttransform1,
                [seq(hilbertpoint(j, k − 1), j = 1 .. 2^(2×k − 1))])[
                i − 2^(2×k − 1)]
            else
                if i ≤ 3×2^(2×k − 1) then map(hilberttransform2,
                    [seq(hilbertpoint(j, k − 1), j = 1 .. 2^(2×k − 1))])[
                    i − 2^(2×k)]
                else map(hilberttransform3,
                    [seq(hilbertpoint(j, k − 1), j = 1 .. 2^(2×k − 1))])[
                    i − 3×2^(2×k − 1)]
                end if
            end if
        end if
    end if
end proc
```

One can compare this algorithm with the algorithm for the Peano points, and note the great similarities. We start with a small set of points, then we basically divide the unit square into quadrants, and apply transformations to our first set of points to get some sort of image of them into each of the subdivisions of the square. We then proceed inductively. Thus, at each **k**, hilbertpoint defines a sequence of points from applying transformations to the set of points at the **k**-1th step. We first start with the set I manually defined as **hilbertpoints1**. For the curious, let us list these points for some **k**:

```
> hilbertpointset:=(k)-
>[seq(hilbertpoint(i,k),i=1..(2^(2*k+1)))];
```

$$hilbertpointset := k \rightarrow [\mathrm{seq}(\mathrm{hilbertpoint}(i, k), i = 1 .. 2^{(2k+1)})]$$

```
> hilbertpointset(1);
```

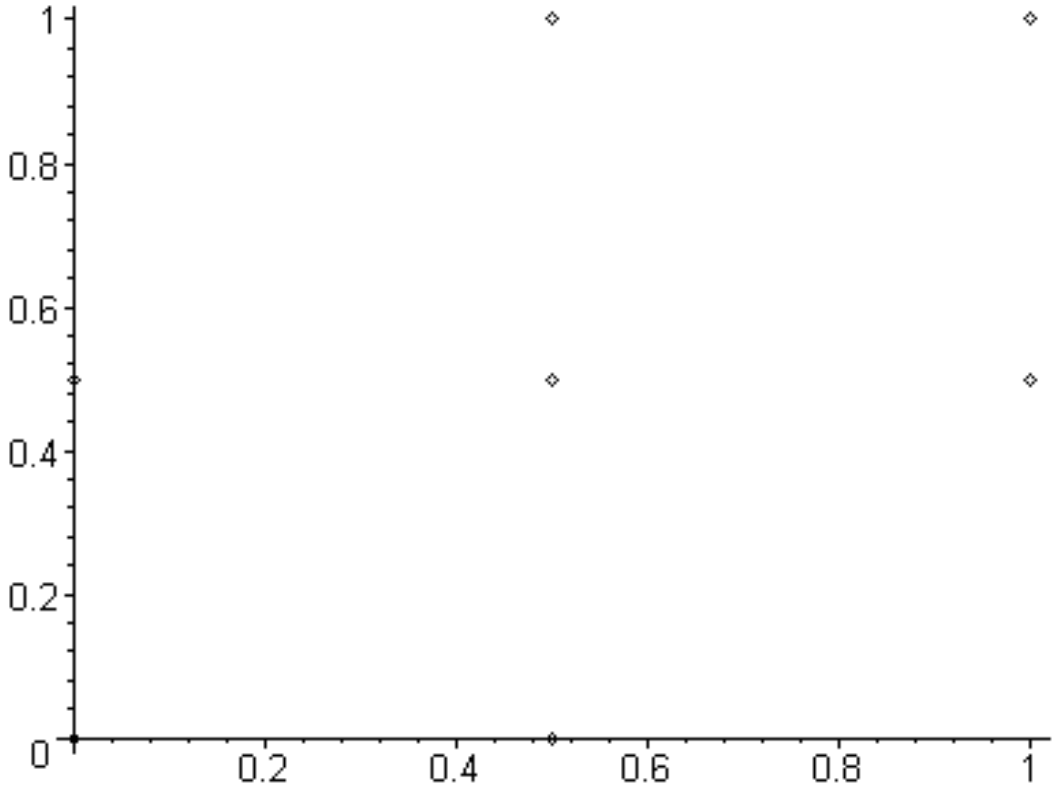
$$\left[[0, 0], \left[\frac{1}{2}, \frac{1}{2}\right], \left[0, \frac{1}{2}\right], \left[\frac{1}{2}, 1\right], \left[\frac{1}{2}, 0\right], \left[1, \frac{1}{2}\right], \left[\frac{1}{2}, \frac{1}{2}\right], [1, 1]\right]$$

```
> hilbertpointset(2);
```

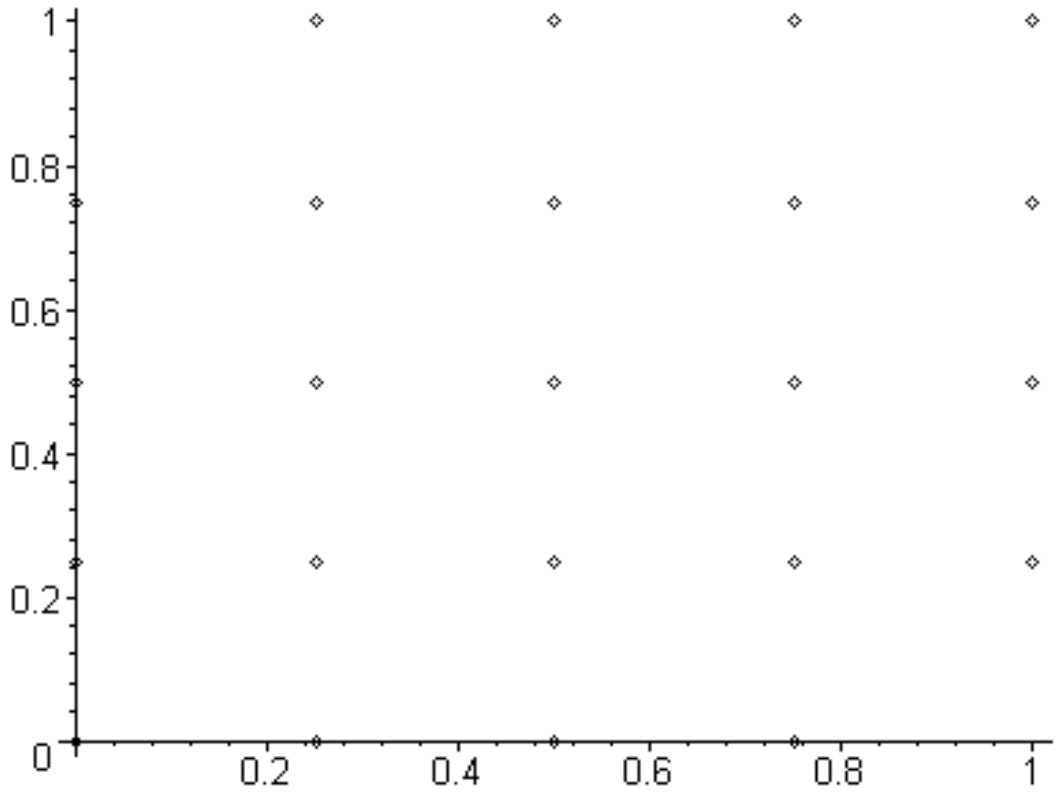
$$\left[[0,0],\left[\frac{1}{4},\frac{1}{4}\right],\left[0,\frac{1}{4}\right],\left[\frac{1}{4},\frac{1}{2}\right],\left[\frac{1}{4},0\right],\left[\frac{1}{2},\frac{1}{4}\right],\left[\frac{1}{4},\frac{1}{4}\right],\left[\frac{1}{2},\frac{1}{2}\right],\left[0,\frac{1}{2}\right],\left[\frac{1}{4},\frac{3}{4}\right],\left[0,\frac{3}{4}\right],\left[\frac{1}{4},1\right],\right.$$
$$\left[\frac{1}{4},\frac{1}{2}\right],\left[\frac{1}{2},\frac{3}{4}\right],\left[\frac{1}{4},\frac{3}{4}\right],\left[\frac{1}{2},1\right],\left[\frac{1}{2},0\right],\left[\frac{3}{4},\frac{1}{4}\right],\left[\frac{1}{2},\frac{1}{4}\right],\left[\frac{3}{4},\frac{1}{2}\right],\left[\frac{3}{4},0\right],\left[1,\frac{1}{4}\right],\left[\frac{3}{4},\frac{1}{4}\right],$$
$$\left.\left[1,\frac{1}{2}\right],\left[\frac{1}{2},\frac{1}{2}\right],\left[\frac{3}{4},\frac{3}{4}\right],\left[\frac{1}{2},\frac{3}{4}\right],\left[\frac{3}{4},1\right],\left[\frac{3}{4},\frac{1}{2}\right],\left[1,\frac{3}{4}\right],\left[\frac{3}{4},\frac{3}{4}\right],[1,1]\right]$$

The first point is always (0,0) while the last is always (1,1). Also, let us plot some of the points:

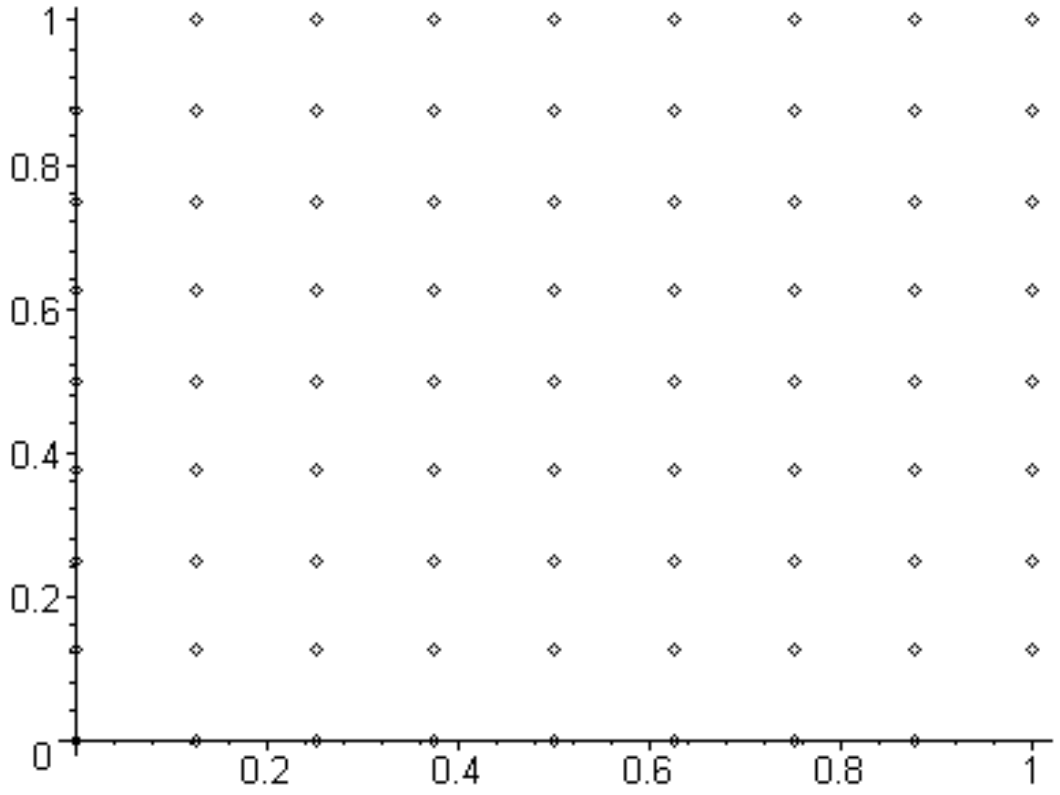
```
> pointplot(hilbertpointset(1));
```



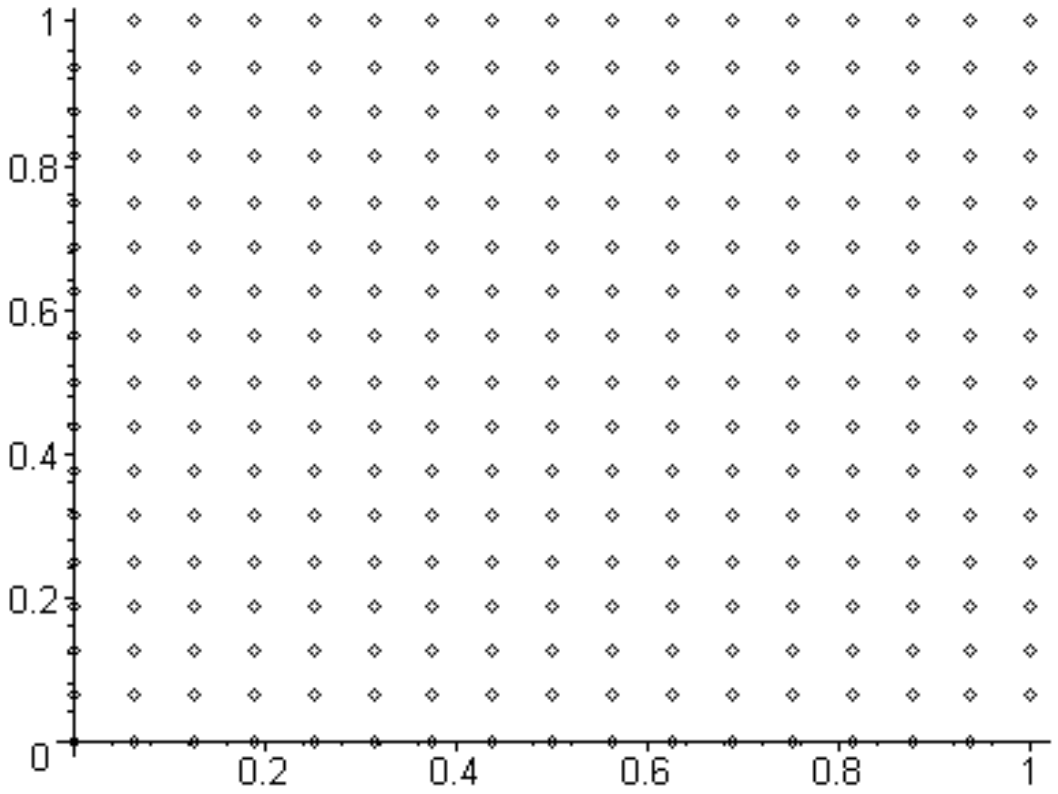

```
> pointplot(hilbertpointset(2));
```




```
> pointplot(hilbertpointset(3));
```




```
> pointplot(hilbertpointset(4));
```



Hopefully, one is convinced that as before, the union of all of these points is a dense set. Thus, our Hilbert Curve which will be continuous and which will pass through all of these points, will hence be onto [0,1]x[0,1].

Let us now define our sequence of curves leading up to the Hilbert Curve. As before, we define our curves in a piecewise manner:

```
> hilbertcurvepart:=(t,i,k)->(hilbertpoint(i+1,k)-
hilbertpoint(i,k))*t+hilbertpoint(i,k);
```

$$hilbertcurvepart := (t, i, k) \to (\text{hilbertpoint}(i+1, k) - \text{hilbertpoint}(i, k))\, t + \text{hilbertpoint}(i, k)$$

So that then we can employ the do-loop method to define a sequence of functions that are piecewise linear:

```
> hilbertcurve:=proc(t,k)
if t=1
then [1,1] else
for i from 1 to ((2^(2*k+1))-1) do
if ((i-1)/((2^(2*k+1))-1)) <= t and t < (i/((2^(2*k+1))-1))
then
hilbertcurvepart((t*((2^(2*k+1))-1))+1-i,i,k); break; else
fi; od;
fi; end;
Warning, `i` is implicitly declared local to procedure `hilbertcurve`
```

```
hilbertcurve := proc(t, k)
local i;
    if t = 1 then [1, 1]
    else for i to 2^(2×k + 1) − 1 do
            if (i − 1)/(2^(2×k + 1) − 1) ≤ t and t < i/(2^(2×k + 1) − 1) then
                hilbertcurvepart(t×(2^(2×k + 1) − 1) + 1 − i, i, k); break end proc
            else
            end if
        end do
    end if
```

So, using:

```
> hilbertcurve1:=(t,k)->hilbertcurve(t,k)[1];
```

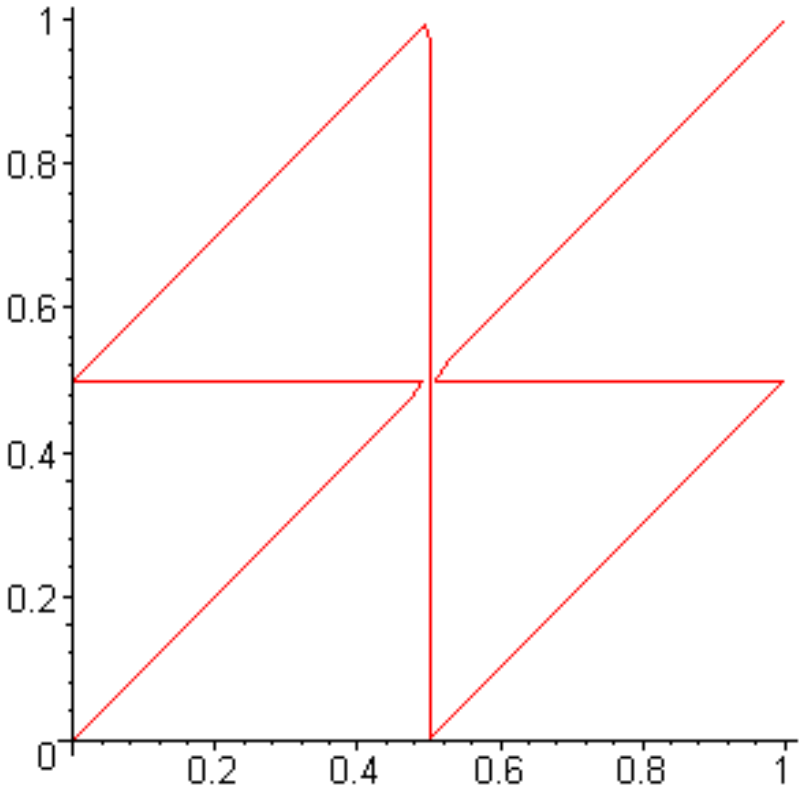
$$hilbertcurve1 := (t, k) \rightarrow \text{hilbertcurve}(t, k)_1$$

```
> hilbertcurve2:=(t,k)->hilbertcurve(t,k)[2];
```

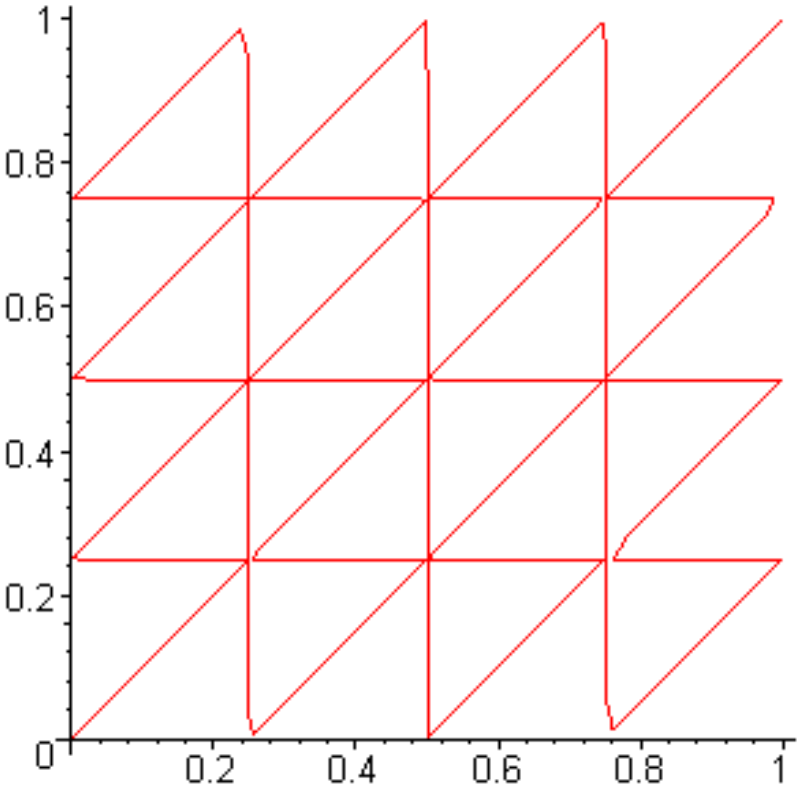
$$hilbertcurve2 := (t, k) \rightarrow \text{hilbertcurve}(t, k)_2$$

We can now finally present what our Hilbert Curves look like:

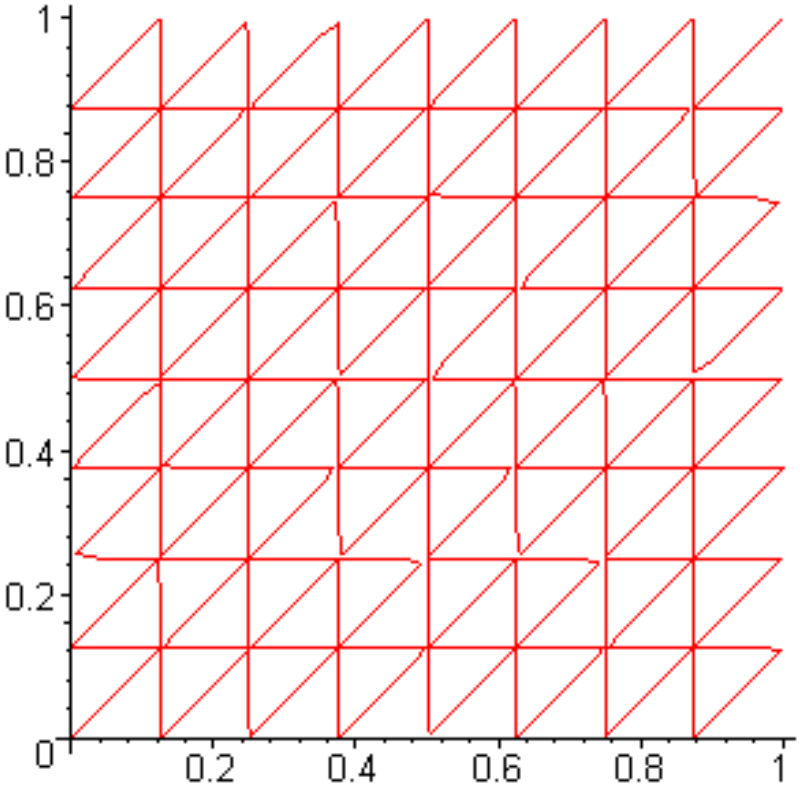
```
> plot(['hilbertcurve1(t,1)','hilbertcurve2(t,1)',t=0..1]);
```



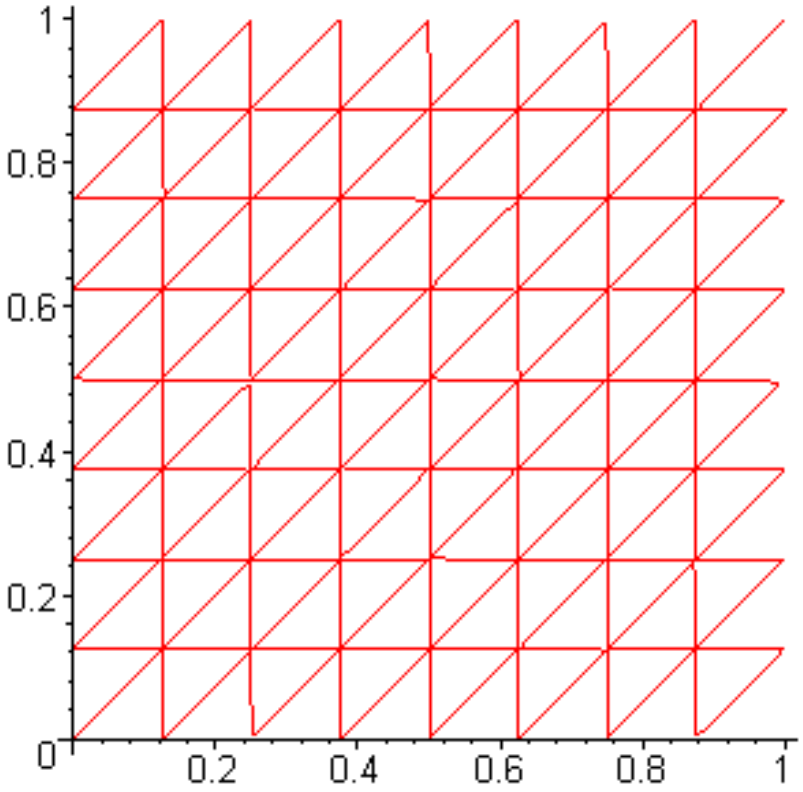

```
> plot(['hilbertcurve1(t,2)','hilbertcurve2(t,2)',t=0..1]);
```



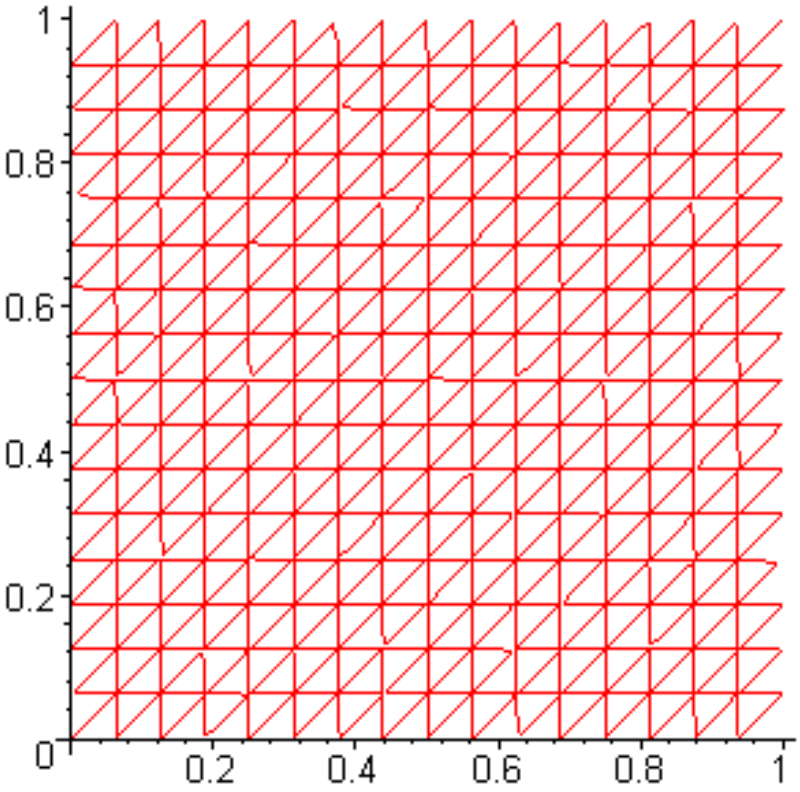

```
> plot(['hilbertcurve1(t,3)','hilbertcurve2(t,3)',t=0..1],numpoints=300);
```



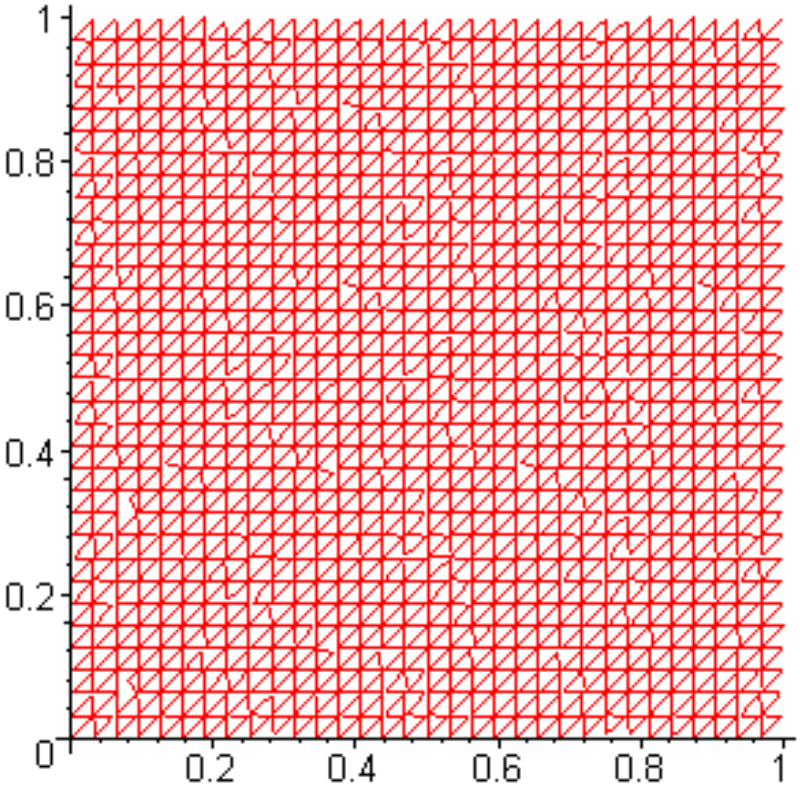
```
>
plot(['hilbertcurve1(t,3)','hilbertcurve2(t,3)',t=0..1],num
points=500);
```




```
>
plot(['hilbertcurve1(t,4)','hilbertcurve2(t,4)',t=0..1],num
points=1000);
```




```
>
plot(['hilbertcurve1(t,5)','hilbertcurve2(t,5)',t=0..1],num
points=2000);
```



The Hilbert Curve is in some sense more efficient than the Peano Curve, for it fills out the unit square out faster.

The setup for the Hilbert Curve is exactly the same as for the Peano Curve. We start by defining a sequence of lists of points, each list including the points from all previous lists, whose union is a dense set in the unit square. We obtain these lists by recursively applying transformations to previous lists. From each list, we define a function by connecting the points in order. Then, our space filling curve will just be the limiting function.
So far what we have done is map the unit segment onto the unit square. Now, we will map the unit segment onto the unit cube. As mentioned before, since we can map the Cantor Set onto the unit segment, then the amazing result is that we can map the Cantor Set onto the unit cube! Let us see an example of a cube filling curve.

Hilbert also presented the idea of a cube-filling curve. Just like the Peano Curve and Hilbert's other curve, this one will also be continuous and nowhere differentiable. Let us now describe this curve.

Hilbert's Cube Filling Curve

Having mastered the tools and techniques used in the construction of the previous curves, programming Hilbert's cube filling curve is routine. As before, we start with an initial list of points, and then define lists of points recursively from this initial list through transformations. Then, we define a sequence of curves from these lists by merely connecting the points in the lists in order. Our Hilbert Cube Curve will then be the limit of this sequence of curves.

We start with the following points in three dimensional space:

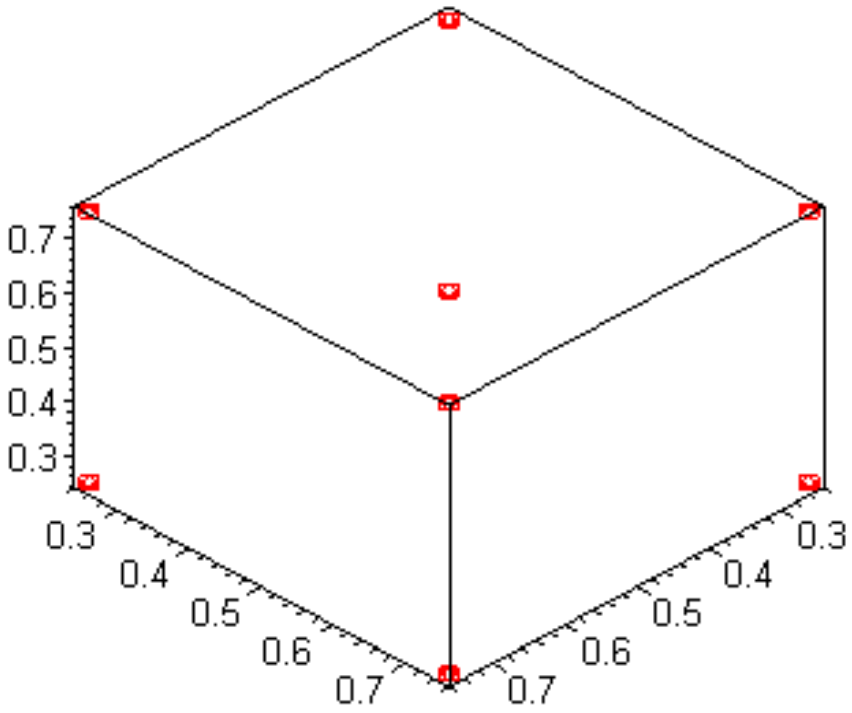
```
> hilbertcubepoints1:=[[1/4,1/4,1/4],[1/4,3/4,1/4],[3/4,3/4,1/4],[3/4,1/4,1/4],[3/4,1/4,3/4],[3/4,3/4,3/4],[1/4,3/4,3/4],[1/4,1/4,3/4]];
```

$$hilbertcubepoints1 := \left[\left[\frac{1}{4},\frac{1}{4},\frac{1}{4}\right],\left[\frac{1}{4},\frac{3}{4},\frac{1}{4}\right],\left[\frac{3}{4},\frac{3}{4},\frac{1}{4}\right],\left[\frac{3}{4},\frac{1}{4},\frac{1}{4}\right],\left[\frac{3}{4},\frac{1}{4},\frac{3}{4}\right],\left[\frac{3}{4},\frac{3}{4},\frac{3}{4}\right],\left[\frac{1}{4},\frac{3}{4},\frac{3}{4}\right],\left[\frac{1}{4},\frac{1}{4},\frac{3}{4}\right]\right]$$

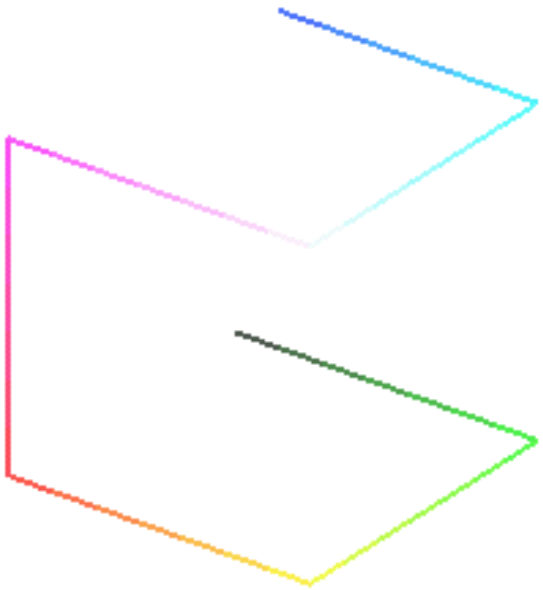
Let us graph them:

```
> with(plots):
Warning, the name changecoords has been redefined
```

```
> pointplot3d([[1/4,1/4,1/4],[1/4,3/4,1/4],[3/4,3/4,1/4],[3/4,1/4,1/4],[3/4,1/4,3/4],[3/4,3/4,3/4],[1/4,3/4,3/4],[1/4,1/4,3/4]],axes=boxed,symbol=circle,symbolsize=24,color=red);
```



As we can see, they occupy the four vertices of a cube. Now, if we connect these four points in the order listed we will get the first function in our sequence, which looks like a clamp:



This curve starts on the bottom far corner, and ends at the top. Now, we introduce eight transformations. Much to my chagrin, these transformations are listed on pages 27-28 of Sagan's "Space Filling Curves." These transformations are nothing more than reflection and rotation followed by scaling and translation. For example, the first transformation takes a point in three space, rotates it about the x-axis by a right angle, reflects it with respect to the xy-plane, and then scales the result by 1/2. The rest are similar in spirit:

```
> hilbertcubetransform1:=(x)->(1/2)*[x[1],x[3],x[2]]+(1/2)*[0,0,0];
```

$$hilbertcubetransform1 := x \rightarrow \left[\frac{1}{2}x_1, \frac{1}{2}x_3, \frac{1}{2}x_2\right]$$

```
> hilbertcubetransform2:=(x)->(1/2)*[x[3],x[2],x[1]]+(1/2)*[0,1,0];
```

$$hilbertcubetransform2 := x \rightarrow \left[\frac{1}{2}x_3, \frac{1}{2}+\frac{1}{2}x_2, \frac{1}{2}x_1\right]$$

```
> hilbertcubetransform3:=(x)->(1/2)*[x[1],x[2],x[3]]+(1/2)*[1,1,0];
```

$$hilbertcubetransform3 := x \rightarrow \left[\frac{1}{2}+\frac{1}{2}x_1, \frac{1}{2}+\frac{1}{2}x_2, \frac{1}{2}x_3\right]$$

```
> hilbertcubetransform4:=(x)->(1/2)*[x[3],-x[1],-x[2]]+(1/2)*[1,1,1];
```

$$hilbertcubetransform4 := x \rightarrow \left[\frac{1}{2}+\frac{1}{2}x_3, \frac{1}{2}-\frac{1}{2}x_1, \frac{1}{2}-\frac{1}{2}x_2\right]$$

```
> hilbertcubetransform5:=(x)->(1/2)*[-x[3],-x[1],x[2]]+(1/2)*[2,1,1];
```

$$hilbertcubetransform5 := x \rightarrow \left[1-\frac{1}{2}x_3, \frac{1}{2}-\frac{1}{2}x_1, \frac{1}{2}+\frac{1}{2}x_2\right]$$

```
> hilbertcubetransform6:=(x)->(1/2)*[x[1],x[2],x[3]]+(1/2)*[1,1,1];
```

$$hilbertcubetransform6 := x \rightarrow \left[\frac{1}{2}+\frac{1}{2}x_1, \frac{1}{2}+\frac{1}{2}x_2, \frac{1}{2}+\frac{1}{2}x_3\right]$$

```
> hilbertcubetransform7:=(x)->(1/2)*[-x[3],x[2],-x[1]]+(1/2)*[1,1,2];
```

$$hilbertcubetransform7 := x \rightarrow \left[\frac{1}{2}-\frac{1}{2}x_3, \frac{1}{2}+\frac{1}{2}x_2, 1-\frac{1}{2}x_1\right]$$

```
> hilbertcubetransform8:=(x)->(1/2)*[x[1],-x[3],-x[2]]+(1/2)*[0,1,2];
```

$$hilbertcubetransform8 := x \rightarrow \left[\frac{1}{2}x_1, \frac{1}{2}-\frac{1}{2}x_3, 1-\frac{1}{2}x_2\right]$$

As before, we will recursively define a sequence of lists by applying these transformations to previous lists. Essentially, we start with the unit cube and the four points in **hilbertcubepoints1**. What will then occur is that we will divide the unti cube

into eight equal cubes. Then, we will take our four points and apply the first transformation to them, the result being that we will get four new points which lie inteirely in one of the small cubes. Thus, we will continue applying transformations to hilbertcubepoints1 until each of the small cubes has a set of four points inside of them. These points will then be the second list of points, to which we will divide the unit cube again into eight pieces and apply our eight transformations again to this new list of points. So, consider the following algorithm:

```
> hilbertcubepoint:=proc(i,k)
option remember;
if k=1 then (hilbertcubepoints1)[i]
else
if i <= 2^(3*k-3) then
map(hilbertcubetransform1,[seq(hilbertcubepoint(j,k-
1),j=1..(2^(3*k-3)))])[i]
else
if i <= 2^(3*k-2) then
map(hilbertcubetransform2,[seq(hilbertcubepoint(j,k-
1),j=1..(2^(3*k-3)))])[i-(2^(3*k-3))]
else
if i <= 3*(2^(3*k-3)) then
map(hilbertcubetransform3,[seq(hilbertcubepoint(j,k-
1),j=1..(2^(3*k-3)))])[i-(2^(3*k-2))]
else
if i <= 2^(3*k-1) then
map(hilbertcubetransform4,[seq(hilbertcubepoint(j,k-
1),j=1..(2^(3*k-3)))])[i-(3*(2^(3*k-3)))]
else
if i <= 5*(2^(3*k-3)) then
map(hilbertcubetransform5,[seq(hilbertcubepoint(j,k-
1),j=1..(2^(3*k-3)))])[i-(2^(3*k-1))]
else
if i <= 3*(2^(3*k-2)) then
map(hilbertcubetransform6,[seq(hilbertcubepoint(j,k-
1),j=1..(2^(3*k-3)))])[i-(5*(2^(3*k-3)))]
else
if i <= 7*(2^(3*k-3)) then
map(hilbertcubetransform7,[seq(hilbertcubepoint(j,k-
1),j=1..(2^(3*k-3)))])[i-(3*(2^(3*k-2)))]
else
map(hilbertcubetransform8,[seq(hilbertcubepoint(j,k-
1),j=1..(2^(3*k-3)))])[i-(7*(2^(3*k-3)))]
fi; fi; fi; fi; fi; fi; fi; fi; end;
```

```
hilbertcubepoint := proc(i, k)
option remember;
    if k = 1 then hilbertcubepoints1[i]
    else
        if i ≤ 2^(3×k – 3) then map(hilbertcubetransform1,
            [seq(hilbertcubepoint(j, k – 1), j = 1 .. 2^(3×k – 3))])[i]
        else
            if i ≤ 2^(3×k – 2) then map(hilbertcubetransform2,
                [seq(hilbertcubepoint(j, k – 1), j = 1 .. 2^(3×k – 3))])[
                i – 2^(3×k – 3)]
            else
                if i ≤ 3×2^(3×k – 3) then map(hilbertcubetransform3,
                    [seq(hilbertcubepoint(j, k – 1), j = 1 .. 2^(3×k – 3))])[
                    i – 2^(3×k – 2)]
                else
                    if i ≤ 2^(3×k – 1) then map(hilbertcubetransform4,
                        [seq(hilbertcubepoint(j, k – 1), j = 1 .. 2^(3×k – 3))])[
                        i – 3×2^(3×k – 3)]
                    else
                        if i ≤ 5×2^(3×k – 3) then map(hilbertcubetransform5, [
                            seq(hilbertcubepoint(j, k – 1), j = 1 .. 2^(3×k – 3))
                            ])[i – 2^(3×k – 1)]
                        else
                            if i ≤ 3×2^(3×k – 2) then map(
                                hilbertcubetransform6, [seq(
                                hilbertcubepoint(j, k – 1), j = 1 .. 2^(3×k – 3))
                                ])[i – 5×2^(3×k – 3)]
                            else
                                if i ≤ 7×2^(3×k – 3) then map(
                                    hilbertcubetransform7, [seq(
                                    hilbertcubepoint(j, k – 1),
                                    j = 1 .. 2^(3×k – 3))])[
                                    i – 3×2^(3×k – 2)]
                                else map(hilbertcubetransform8, [seq(
                                    hilbertcubepoint(j, k – 1),
                                    j = 1 .. 2^(3×k – 3))])[
                                    i – 7×2^(3×k – 3)]
                                end if
                            end if
                        end if
```

```
                    end if  end proc
                end if
            end if
        end if
    end if
```

Define:

```
> hilbertcubepointset:=(k)-
>[seq(hilbertcubepoint(i,k),i=1..(2^(3*k)))];
```

$$hilbertcubepointset := k \to [\,\mathrm{seq}(\mathrm{hilbertcubepoint}(i, k),\ i = 1 \,..\, 2^{(3\,k)})\,]$$

So now let us list our points for some **k**=1,2:

```
> hilbertcubepointset(1);
```

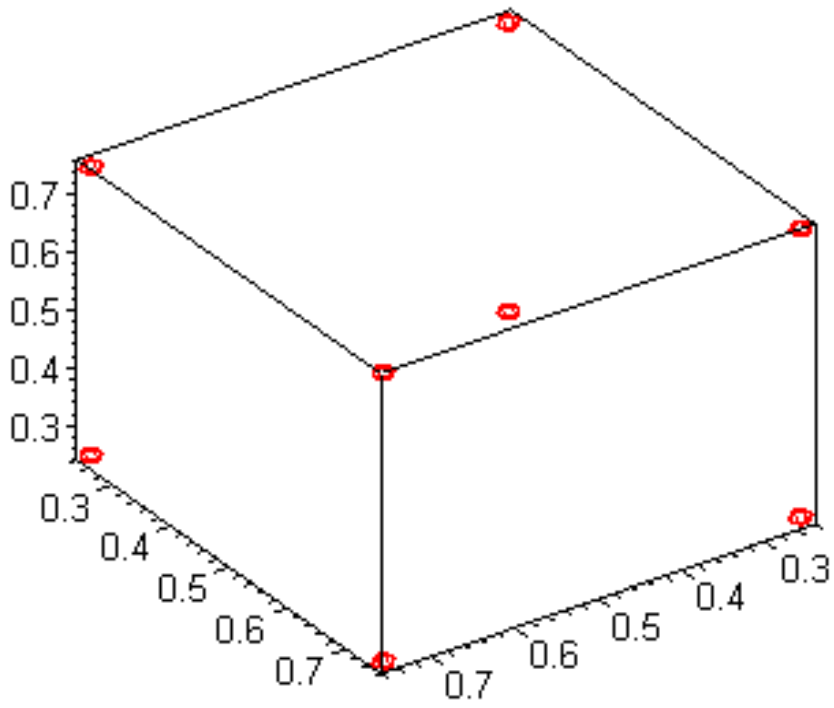
$$\left[\left[\frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right], \left[\frac{1}{4}, \frac{3}{4}, \frac{1}{4}\right], \left[\frac{3}{4}, \frac{3}{4}, \frac{1}{4}\right], \left[\frac{3}{4}, \frac{1}{4}, \frac{1}{4}\right], \left[\frac{3}{4}, \frac{1}{4}, \frac{3}{4}\right], \left[\frac{3}{4}, \frac{3}{4}, \frac{3}{4}\right], \left[\frac{1}{4}, \frac{3}{4}, \frac{3}{4}\right], \left[\frac{1}{4}, \frac{1}{4}, \frac{3}{4}\right]\right]$$

```
> hilbertcubepointset(2);
```

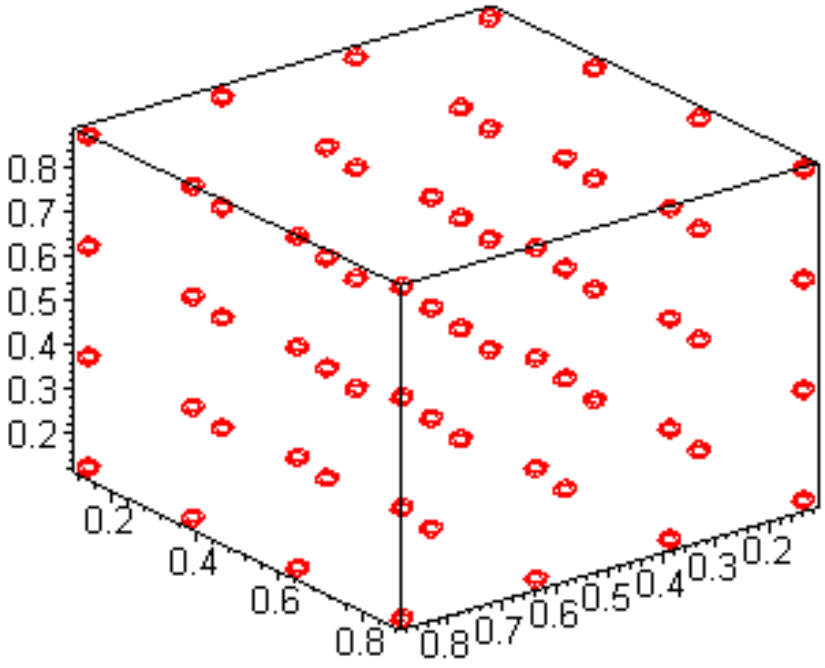
$$\left[\left[\frac{1}{8}, \frac{1}{8}, \frac{1}{8}\right], \left[\frac{1}{8}, \frac{1}{8}, \frac{3}{8}\right], \left[\frac{3}{8}, \frac{1}{8}, \frac{3}{8}\right], \left[\frac{3}{8}, \frac{1}{8}, \frac{1}{8}\right], \left[\frac{3}{8}, \frac{3}{8}, \frac{1}{8}\right], \left[\frac{3}{8}, \frac{3}{8}, \frac{3}{8}\right], \left[\frac{1}{8}, \frac{3}{8}, \frac{3}{8}\right], \left[\frac{1}{8}, \frac{3}{8}, \frac{1}{8}\right],\right.$$
$$\left[\frac{1}{8}, \frac{5}{8}, \frac{1}{8}\right], \left[\frac{1}{8}, \frac{7}{8}, \frac{1}{8}\right], \left[\frac{1}{8}, \frac{7}{8}, \frac{3}{8}\right], \left[\frac{1}{8}, \frac{5}{8}, \frac{3}{8}\right], \left[\frac{3}{8}, \frac{5}{8}, \frac{3}{8}\right], \left[\frac{3}{8}, \frac{7}{8}, \frac{3}{8}\right], \left[\frac{3}{8}, \frac{7}{8}, \frac{1}{8}\right], \left[\frac{3}{8}, \frac{5}{8}, \frac{1}{8}\right],$$
$$\left[\frac{5}{8}, \frac{5}{8}, \frac{1}{8}\right], \left[\frac{5}{8}, \frac{7}{8}, \frac{1}{8}\right], \left[\frac{7}{8}, \frac{7}{8}, \frac{1}{8}\right], \left[\frac{7}{8}, \frac{5}{8}, \frac{1}{8}\right], \left[\frac{7}{8}, \frac{5}{8}, \frac{3}{8}\right], \left[\frac{7}{8}, \frac{7}{8}, \frac{3}{8}\right], \left[\frac{5}{8}, \frac{7}{8}, \frac{3}{8}\right], \left[\frac{5}{8}, \frac{5}{8}, \frac{3}{8}\right],$$
$$\left[\frac{5}{8}, \frac{3}{8}, \frac{3}{8}\right], \left[\frac{5}{8}, \frac{3}{8}, \frac{1}{8}\right], \left[\frac{5}{8}, \frac{1}{8}, \frac{1}{8}\right], \left[\frac{5}{8}, \frac{1}{8}, \frac{3}{8}\right], \left[\frac{7}{8}, \frac{1}{8}, \frac{3}{8}\right], \left[\frac{7}{8}, \frac{1}{8}, \frac{1}{8}\right], \left[\frac{7}{8}, \frac{3}{8}, \frac{1}{8}\right], \left[\frac{7}{8}, \frac{3}{8}, \frac{3}{8}\right],$$
$$\left[\frac{7}{8}, \frac{3}{8}, \frac{5}{8}\right], \left[\frac{7}{8}, \frac{3}{8}, \frac{7}{8}\right], \left[\frac{7}{8}, \frac{1}{8}, \frac{7}{8}\right], \left[\frac{7}{8}, \frac{1}{8}, \frac{5}{8}\right], \left[\frac{5}{8}, \frac{1}{8}, \frac{5}{8}\right], \left[\frac{5}{8}, \frac{1}{8}, \frac{7}{8}\right], \left[\frac{5}{8}, \frac{3}{8}, \frac{7}{8}\right], \left[\frac{5}{8}, \frac{3}{8}, \frac{5}{8}\right],$$
$$\left[\frac{5}{8}, \frac{5}{8}, \frac{5}{8}\right], \left[\frac{5}{8}, \frac{7}{8}, \frac{5}{8}\right], \left[\frac{7}{8}, \frac{7}{8}, \frac{5}{8}\right], \left[\frac{7}{8}, \frac{5}{8}, \frac{5}{8}\right], \left[\frac{7}{8}, \frac{5}{8}, \frac{7}{8}\right], \left[\frac{7}{8}, \frac{7}{8}, \frac{7}{8}\right], \left[\frac{5}{8}, \frac{7}{8}, \frac{7}{8}\right], \left[\frac{5}{8}, \frac{5}{8}, \frac{7}{8}\right],$$
$$\left[\frac{3}{8}, \frac{5}{8}, \frac{7}{8}\right], \left[\frac{3}{8}, \frac{7}{8}, \frac{7}{8}\right], \left[\frac{3}{8}, \frac{7}{8}, \frac{5}{8}\right], \left[\frac{3}{8}, \frac{5}{8}, \frac{5}{8}\right], \left[\frac{1}{8}, \frac{5}{8}, \frac{5}{8}\right], \left[\frac{1}{8}, \frac{7}{8}, \frac{5}{8}\right], \left[\frac{1}{8}, \frac{7}{8}, \frac{7}{8}\right], \left[\frac{1}{8}, \frac{5}{8}, \frac{7}{8}\right],$$
$$\left.\left[\frac{1}{8}, \frac{3}{8}, \frac{7}{8}\right], \left[\frac{1}{8}, \frac{3}{8}, \frac{5}{8}\right], \left[\frac{3}{8}, \frac{3}{8}, \frac{5}{8}\right], \left[\frac{3}{8}, \frac{3}{8}, \frac{7}{8}\right], \left[\frac{3}{8}, \frac{1}{8}, \frac{7}{8}\right], \left[\frac{3}{8}, \frac{1}{8}, \frac{5}{8}\right], \left[\frac{1}{8}, \frac{1}{8}, \frac{5}{8}\right], \left[\frac{1}{8}, \frac{1}{8}, \frac{7}{8}\right]\right]$$

Also, let's plot the sets for several **k**:

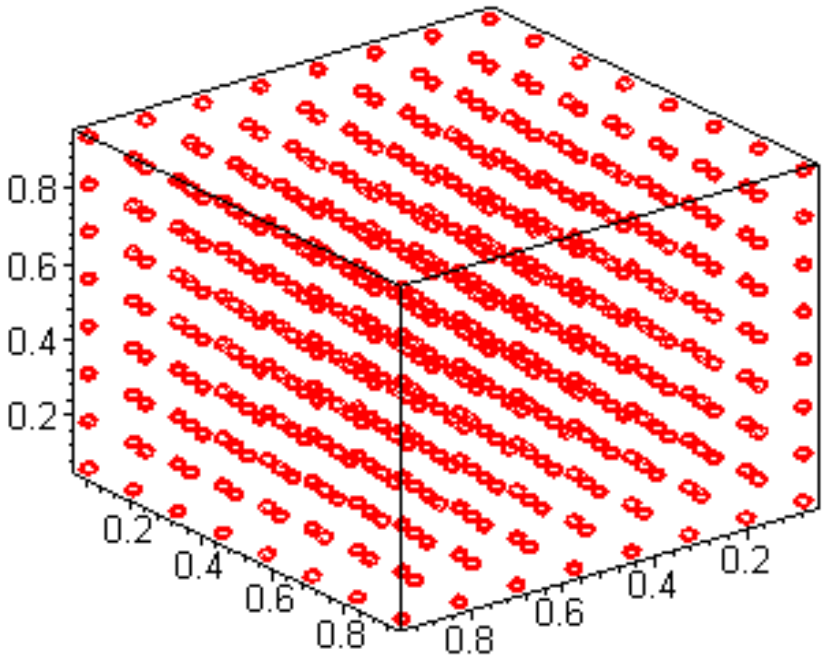
```
>
pointplot3d(hilbertcubepointset(1),axes=boxed,symbol=circle
,symbolsize=24,color=red);
```



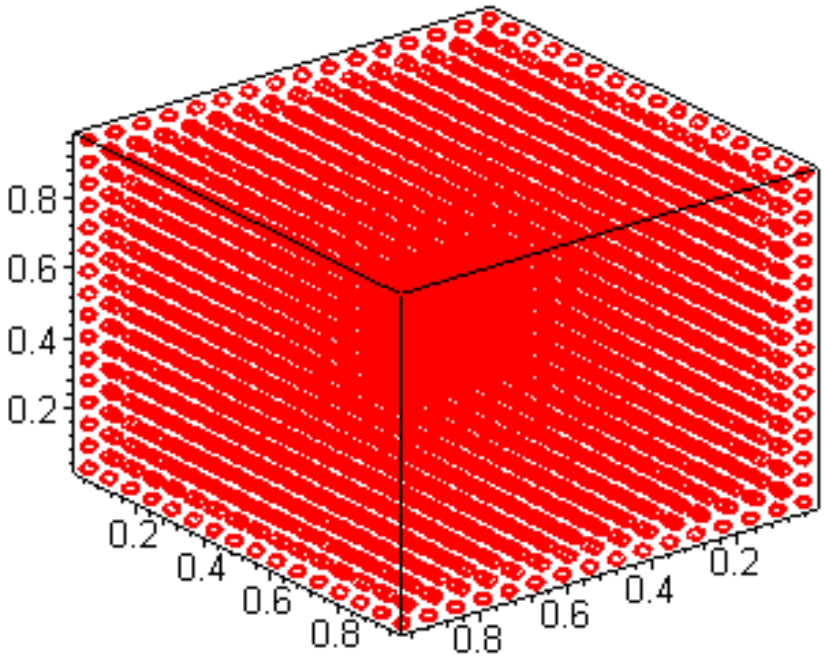
>
```
pointplot3d(hilbertcubepointset(2),axes=boxed,symbol=circle
,symbolsize=24,color=red);
```



```
>
pointplot3d(hilbertcubepointset(3),axes=boxed,symbol=circle
,symbolsize=18,color=red);
```




```
>
pointplot3d(hilbertcubepointset(4),axes=boxed,symbol=circle
,symbolsize=18,color=red);
```



As before, these points are starting to get dense in the unit cube, and so we expect that our Hilbert Cube Curve will be continuous. Thus, let us now program our sequence of functions, again, simply by connecting the above points in order:

```
> hilbertcubecurvepart:=(t,i,k)->(hilbertcubepoint(i+1,k)-
hilbertcubepoint(i,k))*t+hilbertcubepoint(i,k);
```

hilbertcubecurvepart := $(t, i, k) \rightarrow$

$(\text{hilbertcubepoint}(i+1, k) - \text{hilbertcubepoint}(i, k))\, t + \text{hilbertcubepoint}(i, k)$

```
> hilbertcubecurve:=proc(t,k)
if t=0
then hilbertcubepoint(1,k) else
for i from 1 to ((2^(3*k))-1) do
if ((i-1)/((2^(3*k))-1)) < t and t <= (i/((2^(3*k))-1))
then
hilbertcubecurvepart((t*((2^(3*k))-1))+1-i,i,k); break;
else fi; od;
fi; end;
Warning, `i` is implicitly declared local to procedure
`hilbertcubecurve`
```

hilbertcubecurve := **proc** (t, k)

local i;

if $t = 0$ **then** hilbertcubepoint$(1, k)$

else for i **to** $2\text{^}(3\times k) - 1$ **do**

if $(i-1)/(2\text{^}(3\times k) - 1) < t$ **and** $t \le i/(2\text{^}(3\times k) - 1)$ **then**

```
                hilbertcubecurvepart(t×(2^(3×k)−1)+1−i, i, k); break   end proc
            else
            end if
        end do
    end if
```

Finally, lets draw the curves:

```
> hilbertcubecurve1:=(t,k)->hilbertcubecurve(t,k)[1];
```

$$hilbertcubecurve1 := (t, k) \rightarrow \text{hilbertcubecurve}(t, k)_1$$

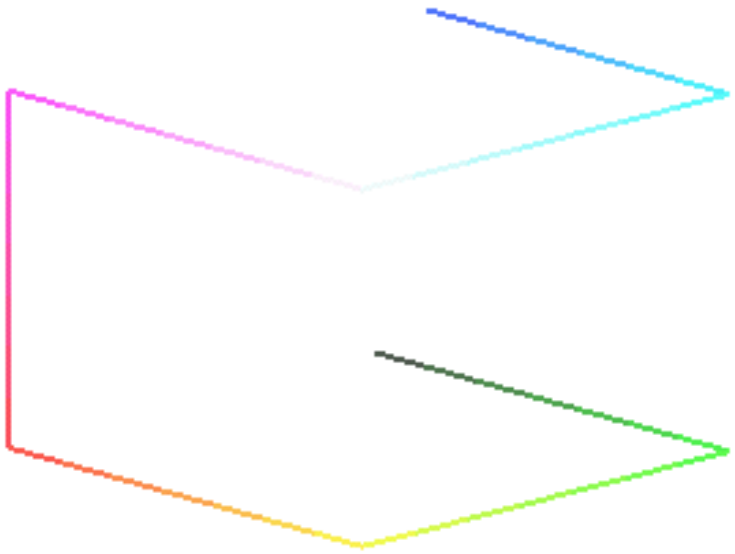
```
> hilbertcubecurve2:=(t,k)->hilbertcubecurve(t,k)[2];
```

$$hilbertcubecurve2 := (t, k) \rightarrow \text{hilbertcubecurve}(t, k)_2$$

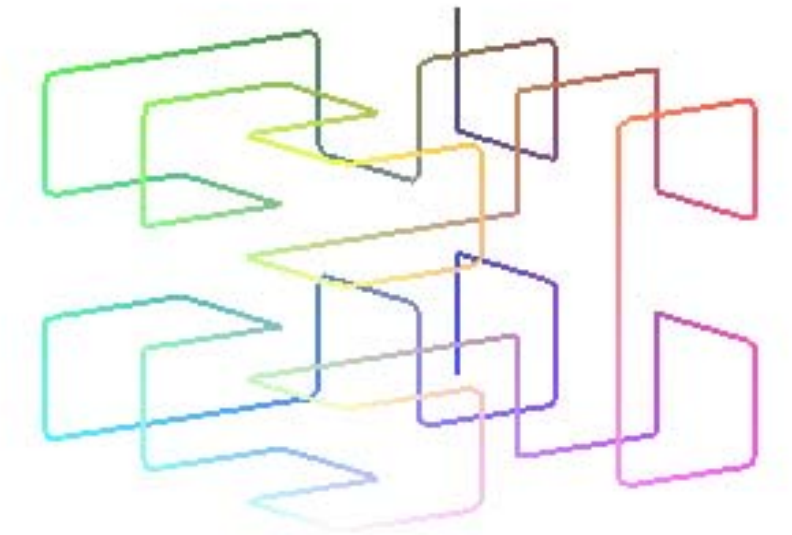
```
> hilbertcubecurve3:=(t,k)->hilbertcubecurve(t,k)[3];
```

$$hilbertcubecurve3 := (t, k) \rightarrow \text{hilbertcubecurve}(t, k)_3$$

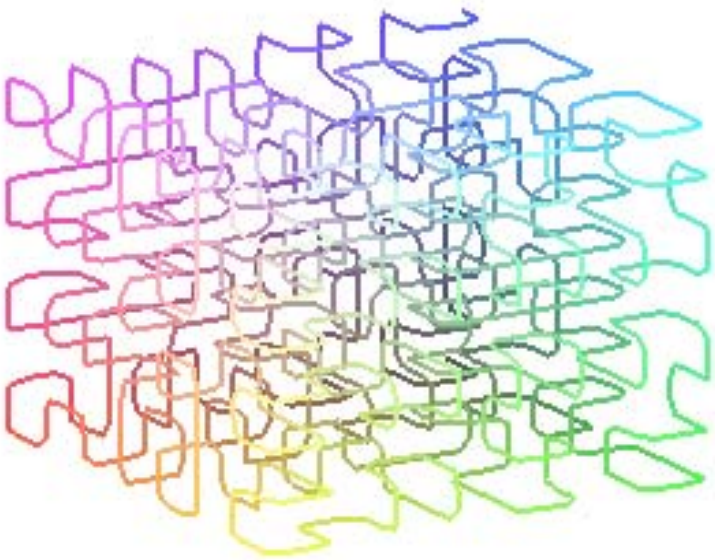
```
>
spacecurve(['hilbertcubecurve1(t,1)','hilbertcubecurve2(t,1
)','hilbertcubecurve3(t,1)'],t=0..1,thickness=2);
```



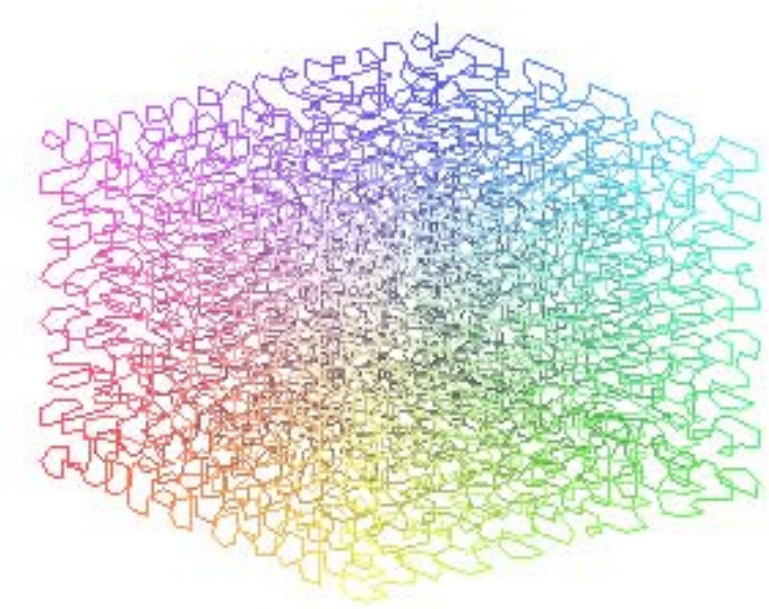
```
>
spacecurve(['hilbertcubecurve1(t,2)','hilbertcubecurve2(t,2
)','hilbertcubecurve3(t,2)'],t=0..1,thickness=2,numpoints=5
00);
```



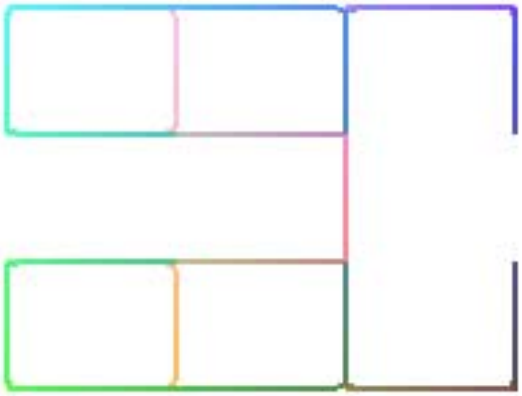
```
>
spacecurve(['hilbertcubecurve1(t,3)','hilbertcubecurve2(t,3
)','hilbertcubecurve3(t,3)'],t=0..1,thickness=2,numpoints=1
200);
```

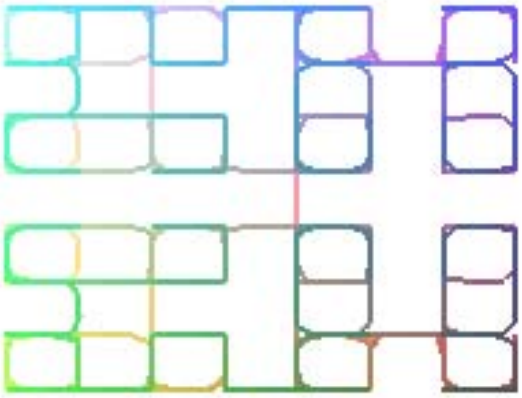


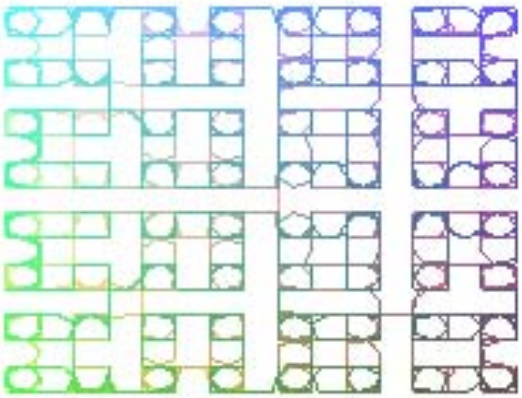
```
>
spacecurve(['hilbertcubecurve1(t,4)','hilbertcubecurve2(t,4
)','hilbertcubecurve3(t,4)'],t=0..1,thickness=1,numpoints=5
000);
```



Hopefully we are convinced that the Hilbert Cube Curve will actually fill out the unit cube. Like the Peano Curve and the Hilbert Space Curve, this curve will also be continuous, nowhere differentiable, and will fail to be injective. However, unlike the Peano Curve and the othe Hilbert Curve, each sequence of curves leading up to the Hilbert Cube Curve does not fail to be injective. There are many interesting patterns in the Hilbert Cube Curve construction. Indeed, by looking at a particular face in this sequence, where the starting point and the enpoint are on the front face and to the left::







Now, think of the above graphs as being the graphs of a sequence curves from [0,1] into the unit square. Then we can see that these curve will converge to a curve that lies on a face mapped out by the Hilbert Cube Curve, and as such, must be onto the unit square.

Thus, the Hilbert Cube Curve has given us another space filling curve.
We have thus managed to map the unit interval onto the unit square and cube, and with some thinking one can see how we can map the cantor set on the unit cube and square. Much more is possible, in fact Felix Hausdorff showed that every compact in **R^n** is a continuous image of some function defined on the Cantor Set. The proof is by construction, and by getting a general feel for these space filling curves we can see that this construction will have to be a recursive one.

The three last examples have all been attacked through the same method. We start with a small set of data which we input by hand, and then we recursively define what we need by applying transformations to this small set. Now, we return to the Cantor Set and apply the same method to it.

Cantor Set Revisited

We shall now introduce an alternative way of computing the cantor points. Our original way was a very direct approach, where the cantor steps at the **k**th step directly depended on the cantor points at the step **k**-1. However, we can get another algorithm for the cantor points by defining the cantor points at the first step by hand and then applying two transformations to it recursively. So, define:

```
> fractalcantorpoints1:=[0,1/3,2/3,1];
```

$$fractalcantorpoints1 := \left[0, \frac{1}{3}, \frac{2}{3}, 1\right]$$

Now, the idea is that to get the cantor points at the step **k**, we get the cantor points of the previous step, shrink then by a third, and then make two copies. We draw one copy on the first third of the unit segment, and the second copy on the last third. Thus, define:

```
> cantortransform1:=x->(1/3)*x;
```

$$cantortransform1 := x \rightarrow \frac{1}{3} x$$

```
> cantortransform2:=x->(1/3)*x+(2/3);
```

$$cantortransform2 := x \rightarrow \frac{1}{3} x + \frac{2}{3}$$

Then our second algorithm for the cantor points, which we call **fractalcantorpoint** for its use of mirroring, is given by:

```
> fractalcantorpoint:=proc(i,k)
option remember;
if k=1 then fractalcantorpoints1[i]
else
if i <= 2^k then
(map(cantortransform1,[seq(fractalcantorpoint(j,k-
1),j=1..2^(k))]))[i];
else (map(cantortransform2,[seq(fractalcantorpoint(j,k-
1),j=1..2^(k))]))[i-2^k];
fi ; fi ; end ;
```

```
fractalcantorpoint := proc(i, k)
option remember;
    if k = 1 then fractalcantorpoints1[i]
    else
        if i ≤ 2^k then
            map(cantortransform1, [seq(fractalcantorpoint(j, k − 1), j = 1 .. 2^k)])
            [i]
        else map(cantortransform2, [seq(fractalcantorpoint(j, k − 1), j = 1 .. 2^k)])
            [i − 2^k]
        end if
    end if
end proc
```

Let us list out some points to convince ourselves this algorithm is correct:

```
> fractalcantorsequence:=(k)->
[seq(fractalcantorpoint(i,k),i=1..2^(k+1))];
```

$$fractalcantorsequence := k \to [\,seq(fractalcantorpoint(i, k), i = 1 .. 2^{(k+1)})\,]$$

```
> fractalcantorsequence(1);
```

$$\left[0, \frac{1}{3}, \frac{2}{3}, 1\right]$$

```
> fractalcantorsequence(2);
```

$$\left[0, \frac{1}{9}, \frac{2}{9}, \frac{1}{3}, \frac{2}{3}, \frac{7}{9}, \frac{8}{9}, 1\right]$$

```
> fractalcantorsequence(3);
```

$$\left[0, \frac{1}{27}, \frac{2}{27}, \frac{1}{9}, \frac{2}{9}, \frac{7}{27}, \frac{8}{27}, \frac{1}{3}, \frac{2}{3}, \frac{19}{27}, \frac{20}{27}, \frac{7}{9}, \frac{8}{9}, \frac{25}{27}, \frac{26}{27}, 1\right]$$

```
> fractalcantorsequence(4);
```

$$\left[0, \frac{1}{81}, \frac{2}{81}, \frac{1}{27}, \frac{2}{27}, \frac{7}{81}, \frac{8}{81}, \frac{1}{9}, \frac{2}{9}, \frac{19}{81}, \frac{20}{81}, \frac{7}{27}, \frac{8}{27}, \frac{25}{81}, \frac{26}{81}, \frac{1}{3}, \frac{2}{3}, \frac{55}{81}, \frac{56}{81}, \frac{19}{27}, \frac{20}{27}, \frac{61}{81}, \frac{62}{81}, \frac{7}{9}, \frac{8}{9}, \frac{73}{81}, \frac{74}{81}, \frac{25}{27}, \frac{26}{27}, \frac{79}{81}, \frac{80}{81}, 1\right]$$

At this point one can also rewrite the General Cantor Set algorithm, but lets not be pedantic.

Another point of interest is that of similarity dimension. In each curve we constructed we always started with a sequence of lists of points which were constructed inductively. There were interesting patterns in all of them. As such, there is a way to assign a number to these sequences of sets which rates the complexity of each sequence. An example of this is the similarity dimension, which to put it technically is the ratio of the logarithm of the number of transformations used in the construction of the object, over the logarithm of the inverse of the reduction ratio. So for example, if we inductively construct a list of points where each time we reduce the points by a scale of **r,** and then apply **n**

transformations to these points, then the similarity dimension of our end result is ln(**n**)/ln(1/**r**). As the similarity dimension gets bigger, we say that the object is more complex.

So consider the list of points constructed by peanopoint and hilbertpoint. In each case, we had four transformations and at each step, we shrunk the points from the previous step by 1/2. Thus, our similarity dimension is ln(4)/ln(1/1/2) which is 2. For the hilbertcubepoints, we had eight transformations and a reduction ratio of 1/2, and so one can check that we will get a dimension of 3. Thus, whereas the Hilbert Space Filling Curve and the Peano Curve are just as complex as one another, the Hilbert Cube Curve is more complex than these two, as is to be expected. The Cantor Set has similarity dimension ln(2)/ln(3).

We have thus managed to explore several interesting examples of sets and functions. We first discussed the Cantor Set, an uncountable compact set of measure zero, which we later generalized. After, we explored and approximated the Devil's Staircase, an increasing function whose derivative is zero almost everywhere. After giving an example of a nowhere differentiable but everywhere continuous function, we then explored the subject of space-filling curves. We were able to fill out the unit square and unit cube with continuous but nowhere differentiable functions. Along the way, we solved some key programming issues. We learned two techniques to how to define a sequence of functions which are piecewisely defined. One method was using sums, while another was through do-loops. We also mastered the technique of using transformations to recursively define a sequence of lists.