```
> restart;
```

SpaceTree Representation

Bit representation of the underlying SpaceTree (1=node, 0=leaf):

```
> SPACETREE := [1,0,0,0,0];
```

$$SPACETREE := [1, 0, 0, 0, 0]$$

Pointer to current element in SPACETREE:

```
> STPTR := 0;
```

$$STPTR := 0$$

Construct a uniformly refined tree of given depth:

```
> fullTree := proc(depth::integer)
      local ones, zeros;
      if depth = 0
      then return [0]
      else
         return [ 1, seq( op( fullTree(depth-1) ), k=1..4) ];
      end if;
   end proc;
```

fullTree := **proc**(*depth*::*integer*)

local *ones*, *zeros*;

if *depth* = 0 **then return** [0]

else return [1, seq(op(fullTree(*depth* − 1)), *k* = 1 .. 4)]

end if

end proc

A "Fibonnacci-SpaceTree" (the depth decreases for the two last subtrees):

```
> fibTree := proc(depth::integer)
      local ones, zeros;
      if depth = 0
      then return [0]
      elif depth = 1
      then return [ 1, seq( op( fibTree(depth-1) ), k=1..4) ];
      elif depth = 2
      then return [ 1, seq( op( fibTree(depth-1) ), k=1..2),
   seq( op( fibTree(depth-2) ), k=3..4)];
      else return [ 1, op( fibTree(depth-1) ),  seq( op(
   fibTree(depth-2) ), k=2..3), op( fibTree(depth-3) ) ];
      end if;
   end proc;
```

fibTree := **proc**(*depth*::*integer*)

local *ones*, *zeros*;

if *depth* = 0 **then return** [0]

elif *depth* = 1 **then return** [1, seq(op(fibTree(*depth* − 1)), *k* = 1 .. 4)]

elif *depth* = 2 **then return** [1, seq(op(fibTree(*depth* − 1)), *k* = 1 .. 2),

seq(op(fibTree(*depth* − 2)), *k* = 3 .. 4)]

```
    else return [1, op(fibTree(depth − 1)), seq(op(fibTree(depth − 2)), k = 2 .. 3),
        op(fibTree(depth − 3))]
    end if
end proc
```

Examples:

```
> SPACETREE := fullTree(2);
```

SPACETREE := [1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0]

```
> fibTree(3);
```

[1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]

Expand a leaf into a refined node:

```
> adaptTree := proc(st::list, position::integer)
    if (st[position] = 1) then return st; end if;
    return [ seq(st[i], i=1..position-1), 1, 0,0,0,0,
  seq(st[i], i=position+1..nops(st) ) ];
  end proc;
```

```
adaptTree := proc(st::list, position::integer)
    if st[position] = 1 then return st end if;
    return [seq(st[i], i = 1 .. position − 1), 1, 0, 0, 0, 0,
        seq(st[i], i = position + 1 .. nops(st))]
end proc
```

```
> SPACETREE := adaptTree(fullTree(2), 15);
```

SPACETREE := [1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]

Helper Functions for Turtle Graphics

attach an element to the end of a list (of points):

```
> attach := proc(li, elem)
    # eval(li) as li might be call-by-reference
    return [ op(eval(li)),elem];
  end proc;
```

attach := **proc**(*li*, *elem*) **return** [op(eval(*li*)), *elem*] **end proc**

Initialise "turtle" (start in the center of the unit square):

```
> init_turtle := proc()
    global points, stepsize, TURTLE_x, TURTLE_y;
    points := [];
    stepsize := 1;
    TURTLE_x := 1/2;
    TURTLE_y := 1/2;
  end proc;
```

```
init_turtle := proc()
global points, stepsize, TURTLE_x, TURTLE_y;
    points := [ ]; stepsize := 1; TURTLE_x := 1 / 2; TURTLE_y := 1 / 2
end proc
```

```
> mark := proc()
```

```
      #option trace;
      global points, TURTLE_x, TURTLE_y;
      points := attach(points, [ TURTLE_x, TURTLE_y ] );
   end proc;
```

mark := **proc**()

global *points*, *TURTLE_x*, *TURTLE_y*;

points := attach(*points*, [*TURTLE_x*, *TURTLE_y*])

end proc

```
> up := proc()
      #option trace;
      global stepsize, TURTLE_x, TURTLE_y;
      TURTLE_y := TURTLE_y + stepsize;
   end proc;
```

up := **proc**()

global *stepsize*, *TURTLE_x*, *TURTLE_y*;

TURTLE_y := *TURTLE_y* + *stepsize*

end proc

```
> down := proc()
      #option trace;
      global stepsize, TURTLE_x, TURTLE_y;
      TURTLE_y := TURTLE_y - stepsize;
   end proc;
```

down := **proc**()

global *stepsize*, *TURTLE_x*, *TURTLE_y*;

TURTLE_y := *TURTLE_y* – *stepsize*

end proc

```
> left := proc()
      #option trace;
      global stepsize, TURTLE_x, TURTLE_y;
      TURTLE_x := TURTLE_x - stepsize;
   end proc;
```

left := **proc**()

global *stepsize*, *TURTLE_x*, *TURTLE_y*;

TURTLE_x := *TURTLE_x* – *stepsize*

end proc

```
> right := proc()
      #option trace;
      global stepsize, TURTLE_x, TURTLE_y;
      TURTLE_x := TURTLE_x + stepsize;
   end proc;
```

right := **proc**()

global *stepsize*, *TURTLE_x*, *TURTLE_y*;

TURTLE_x := *TURTLE_x* + *stepsize*

```
end proc
```

```
> fine := proc(xshift::integer, yshift::integer)
      # option trace;
      global stepsize, TURTLE_x, TURTLE_y;
      stepsize := stepsize / 2;
      TURTLE_x := TURTLE_x + xshift*stepsize/2;
      TURTLE_y := TURTLE_y + yshift*stepsize/2;
   end proc;
```

fine := **proc**(*xshift*::*integer*, *yshift*::*integer*)

global *stepsize*, *TURTLE_x*, *TURTLE_y*;

stepsize := *stepsize* / 2;

TURTLE_x := *TURTLE_x* + 1 / 2∗*xshift*∗*stepsize*;

TURTLE_y := *TURTLE_y* + 1 / 2∗*yshift*∗*stepsize*

end proc

```
> coarse := proc(xshift::integer, yshift::integer)
      # option trace;
      global stepsize, TURTLE_x, TURTLE_y;
      TURTLE_x := TURTLE_x + xshift*stepsize/2;
      TURTLE_y := TURTLE_y + yshift*stepsize/2;
      stepsize := stepsize * 2;
   end proc;
```

coarse := **proc**(*xshift*::*integer*, *yshift*::*integer*)

global *stepsize*, *TURTLE_x*, *TURTLE_y*;

TURTLE_x := *TURTLE_x* + 1 / 2∗*xshift*∗*stepsize*;

TURTLE_y := *TURTLE_y* + 1 / 2∗*yshift*∗*stepsize*;

stepsize := 2∗*stepsize*

end proc

Algorithm for the adaptive Hilbert Curve

```
> H := proc()
      # option trace;
      global SPACETREE, STPTR;

      STPTR := STPTR + 1;
      if SPACETREE[STPTR] = 0
      then
         mark();
      else
         # recursive calls to children
         fine(-1,-1);
         A(); up();
         H(); right();
         H(); down();
         B();
```

```
        coarse(-1,1);
      end if;
   end proc:
> A := proc()
      # option trace;
      global SPACETREE, STPTR;

      STPTR := STPTR + 1;
      if SPACETREE[STPTR] = 0
      then
         mark();
      else
         fine(-1, -1);
         H(); right();
         A(); up();
         A(); left();
         C();
         coarse(1,-1);
      end if;
   end proc:
> B := proc()
      # option trace;
      global SPACETREE, STPTR;

      STPTR := STPTR + 1;
      if SPACETREE[STPTR] = 0
      then
         mark();
      else
         fine(1, 1);
         C(); left();
         B(); down();
         B(); right();
         H();
         coarse(-1, 1);
      end if;
   end proc:
> C := proc()
      # option trace;
      global SPACETREE, STPTR;

      STPTR := STPTR + 1;
      if SPACETREE[STPTR] = 0
      then
         mark();
      else
         fine(1, 1);
```

```
        B(); down();
        C(); left();
        C(); up();
        A();
      coarse(1, -1);
    end if;
  end proc:
> Hilbert := proc(st::list)
    global points, SPACETREE, STPTR;
    SPACETREE := st;
    STPTR := 0;
    init_turtle();
    H();
    return eval(points);
  end proc;
```

Hilbert := **proc**(*st*::*list*)

global *points*, *SPACETREE*, *STPTR*;

SPACETREE := *st*; *STPTR* := 0; init_turtle( ); H( ); **return** eval(*points*)

end proc

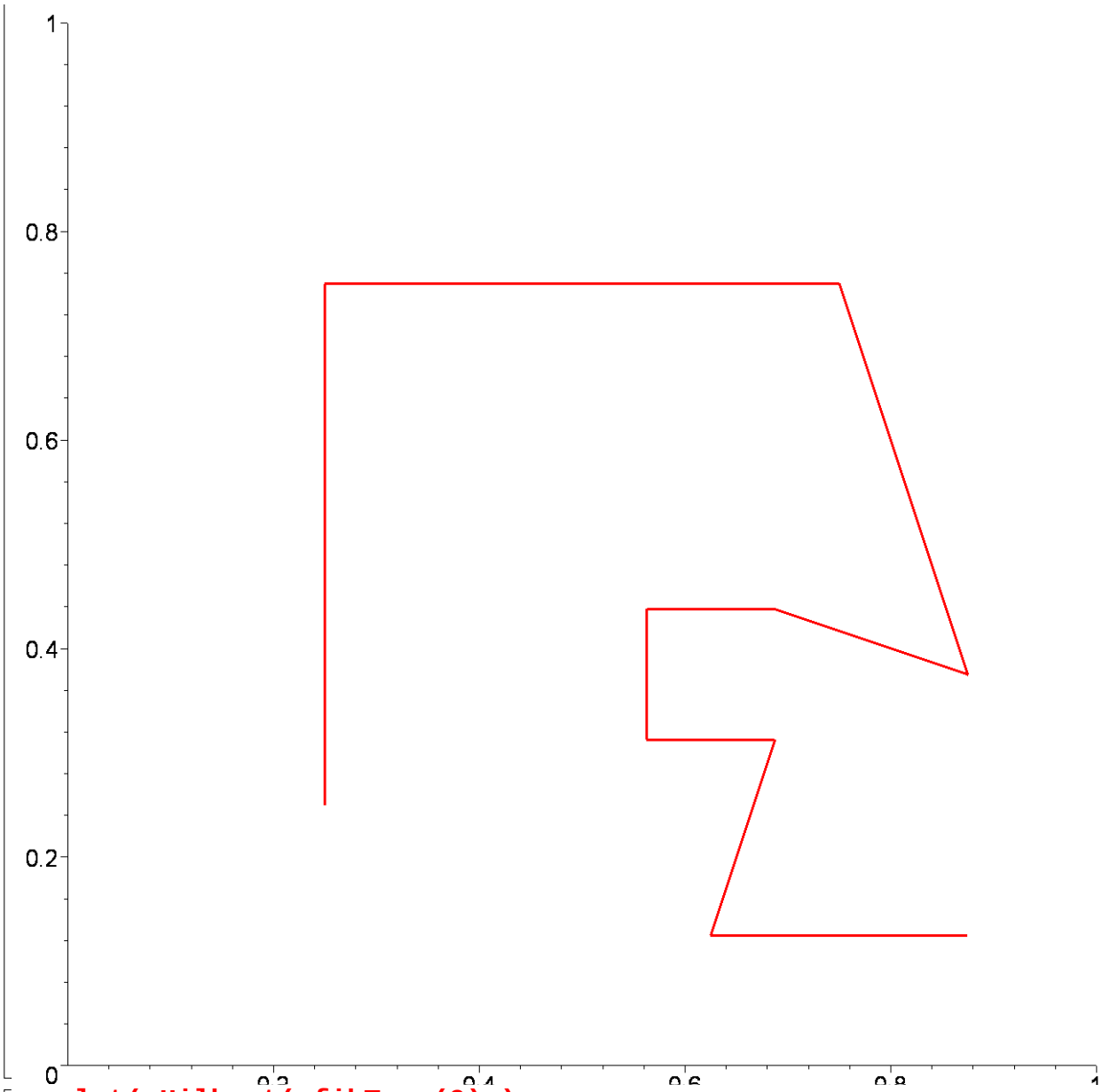
```
> Hilbert( [1,0,0,0,0] );
```

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

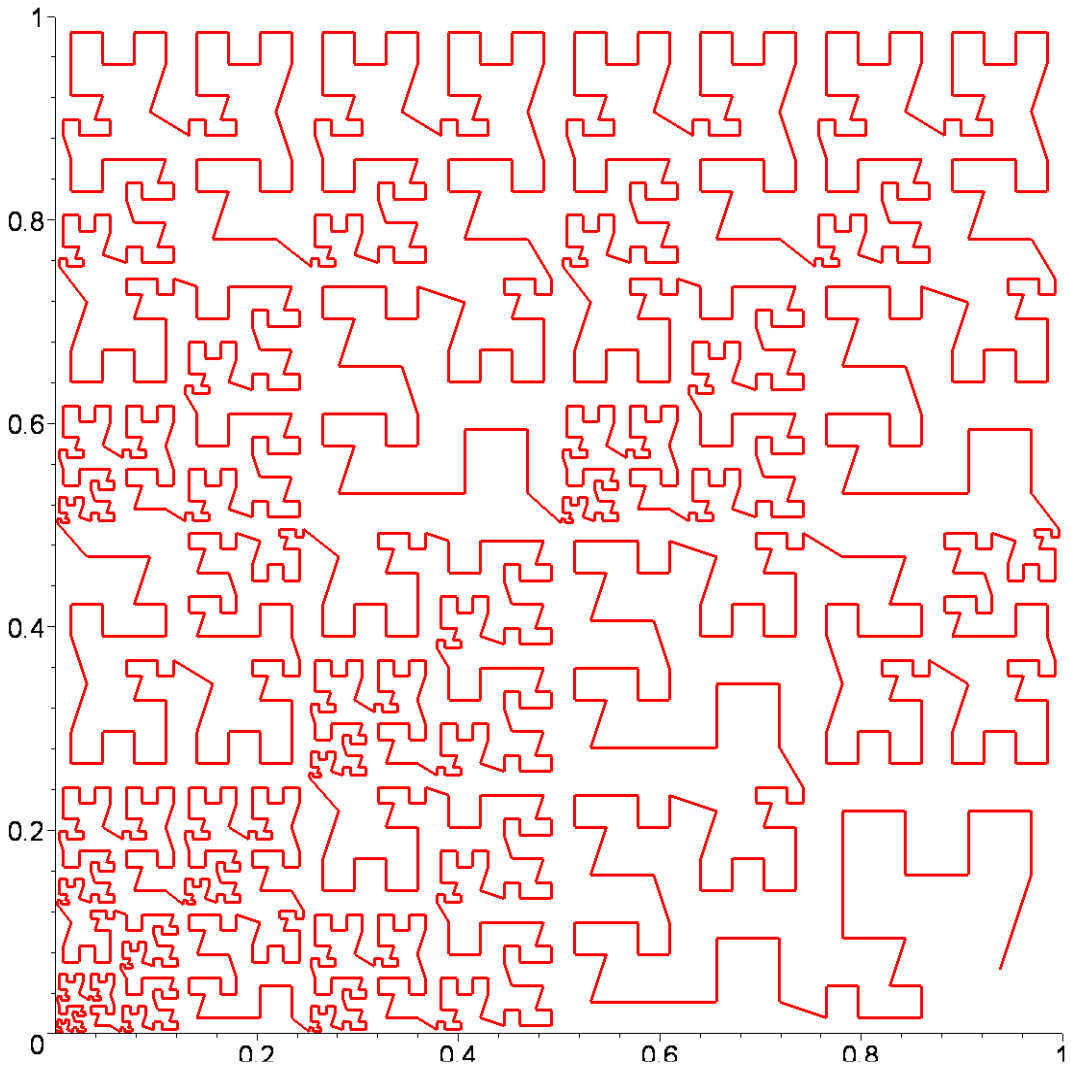
```
> fibTree(2);
```

$$[1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$$

```
> plot( Hilbert( [1,0,0,0,1,0,1,0,0,0,0,0,0] ),
        scaling=CONSTRAINED, thickness=3, view=[0..1, 0..1]);
```




```
> plot( Hilbert( fibTree(9) ),
        scaling=CONSTRAINED, thickness=3, view=[0..1, 0..1]);
```




```
>
>
```

Partitioning of the Hilbert Curve

```
> colours := [black,red, green, yellow, brown, magenta, cyan,
  navy,
                pink, grey, blue, khaki, coral];
```

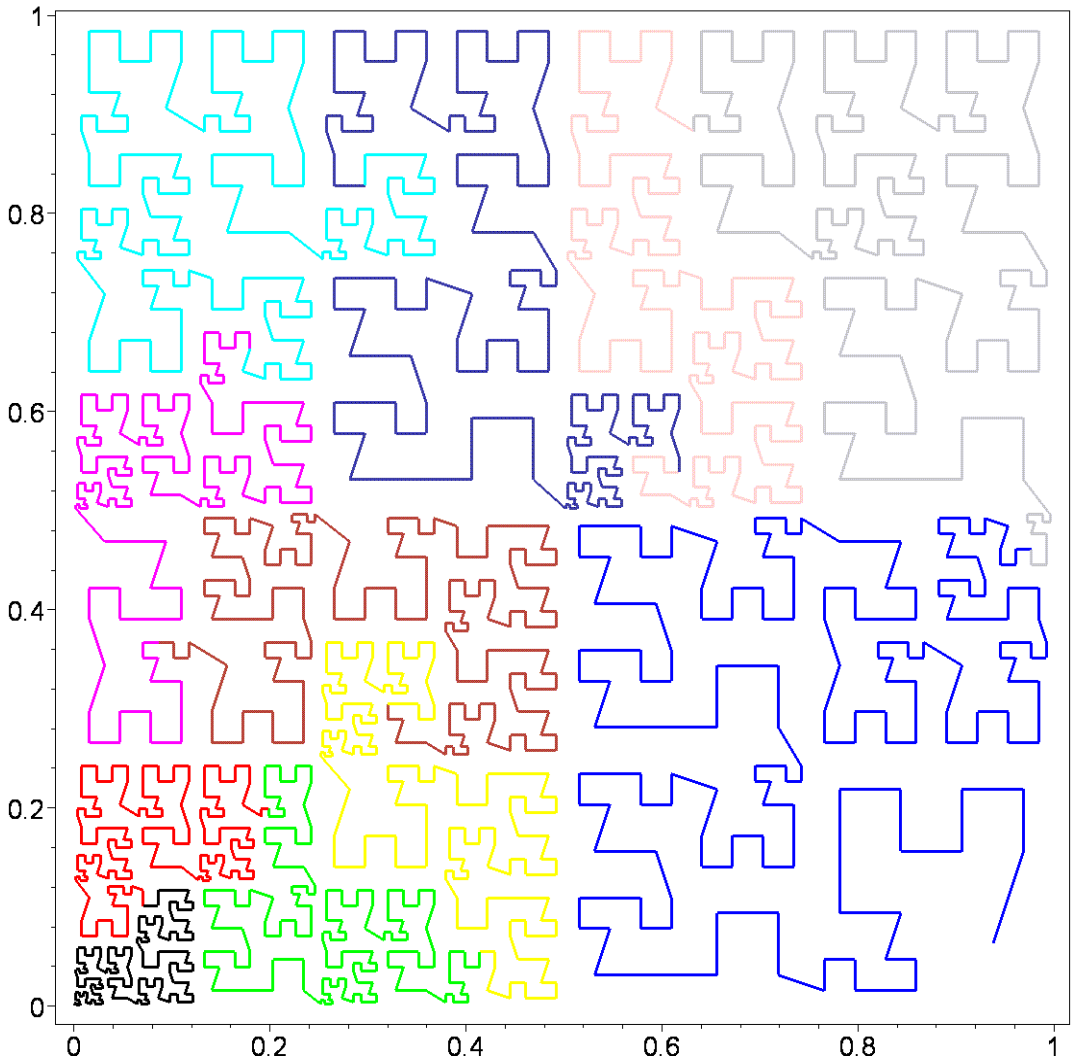
colours :=

[*black*, *red*, *green*, *yellow*, *brown*, *magenta*, *cyan*, *navy*, *pink*, *grey*, *blue*, *khaki*, *coral*]

partition splits the given list pts into number partitions (each partition is again a list)

```
> partition := proc(pts::list, number::posint)
    local parts,i;
    parts := [ pts[ (number-1)*floor(nops(pts)/number)..-1 ]
  ];
    for i from number-1 by -1 to 2 do
       parts := [ pts[
  (i-1)*floor(nops(pts)/number)..i*floor(nops(pts)/number) ],
                  op(parts) ];
    end do;
    parts := [ pts[ 1..floor(nops(pts)/number) ], op(parts)
```

```
    ];
       return parts;
    end proc:
> 
> pts := Hilbert(fibTree(9)):parts := partition(pts,11):
  plot(parts,axes=BOXED, scaling=CONSTRAINED, thickness=3,
  color=colours);
```




```
> 
```