> restart;
with(plots):

\section*{Helper Functions to Draw Hilbert Iterations}

The applied "vertex labelling" algorithm will generate the subsquares of the Hilbert construction as combination of 4 coordinates. Coordinates are represented as a Maple list of 2 elements. Function \texttt{mid4} will return the centre of a subsquare specified by its four corners. Function \texttt{mid2} will return the midpoint between two points, i.e. the centre of the connecting edge.

Function \texttt{attach} will attach a coordinate to a given list of points (for plotting the respective connecting polygonal line). Function \texttt{attachall}, similar to \texttt{attach}, will attach all coordinates specified in the list \texttt{elems}. Function \texttt{mark} attaches a coordinate to the global list points. Function \texttt{markcube} attaches all points required to draw the respective subsquare as a polygonal line.

\begin{verbatim}
mid4 := proc (A::list,B::list,C::list,D::list)
  return (A+B+C+D)/4;
end proc:

mid2 := proc (A::list,B::list)
  return (A+B)/2;
end proc:

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

attachall := proc(li, elems)
  # eval(li) as li might be call-by-reference
  return [ op(eval(li)),op(elems)];
end proc:

mark := proc(vertices::list)
  #option trace;
  global points;
  points := attach(points, mid4( op(vertices) ));
end proc:

markcube := proc(v1::list,v2::list)
  # option trace;
  global cubes;
  cubes := attachall(cubes, [v1,[v1[1],v2[2]],v2,[v2[1],v1[2]],v1,[v1[1],v2[2]],v2]);
end proc:
\end{verbatim}

\section*{Vertex-Labeling Algorithm for the Hilbert Curve}

The functions \texttt{HilbertVL} and \texttt{Hilbert} implement a vertex-labeling algorithm to draw iterations of the 2D Hilbert curve. \texttt{HilbertVL} takes the desired recursion \texttt{depth} and a list of vertices as parameter; the
The `vertices` list contains the 4 coordinates of the subsquare vertices. Their respective order in the list encodes the orientation of the Hilbert curve within the subsquare. The procedure `Hilbert` will call `HilbertVL` to generate Hilbert iterations of specified depth.

```plaintext
HilbertVL := proc(depth::integer, vertices::list)
  if depth = 0
    then mark(vertices)
  else
    HilbertVL( depth-1,
                [ vertices[1],
                  mid2( vertices[1],vertices[4]),
                  mid4( op( vertices) ),
                  mid2( vertices[1],vertices[2]) ] );
    HilbertVL( depth-1,
                [ mid2( vertices[1],vertices[2]),
                  vertices[2],
                  mid2( vertices[2],vertices[3]),
                  mid4( op( vertices) ) ] );
    HilbertVL( depth-1,
                [ mid4( op( vertices) ),
                  mid2( vertices[2],vertices[3]),
                  vertices[3],
                  mid2( vertices[3],vertices[4]) ] );
    HilbertVL( depth-1,
                [ mid2( vertices[3],vertices[4]),
                  mid4( op( vertices) ),
                  mid2( vertices[1],vertices[4]),
                  vertices[4] ] );
  end;
end proc:

Hilbert := proc(depth::integer)
  global points;
  local unitsquare;
  points := [];
  unitsquare := [[0,0],[0,1],[1,1],[1,0]];
  HilbertVL(depth, unitsquare);
  return points;
end proc:

plot( Hilbert( 6 ),
      scaling=CONSTRAINED, thickness=3, view=[0..1, 0..1]);
```
Adaptive Hilbert Curve

In the following, we will generate a Hilbert order on the cells of an adaptive quadtree. The quadtree is represented via a variant of the bitstream encoding: instead of 0/1 bits to represent whether a quadtree node is an inner node or a leaf, we will apply an integer stream, where each number represents the number of nodes contained in its respective subtree. Hence, 1 represents a leaf node, and 5 would represent a quadtree of height 2 with one inner node and four leaves.

The global variable \texttt{SPACETREE} contains such a list of integers (in depth-first traversal order of the quadtree).

The global variable \texttt{STPTR} points to the currently accessed node in \texttt{SPACETREE}.  

Define spacetree representation and stack pointer:

\[ \text{SPACETREE} := [5,1,1,1,1] \]
\[ \text{STPTR} := 0 \]

> fullTree := proc(depth::integer)
>     # generate a full quadtree of given depth
>     local ones, zeros, k, ft;
>     if depth = 0
>         then return [1]
>     else
>         for k from 1 to 4 do
>             ft[k] := fullTree(depth-1);
>         end do;
>         return [ 1+add( ft[k][1], k=1..4 ), seq( op(ft[k]), k=1..4 ) ];
>     end if;
> end proc:

> fullTree(2);
\[ [21, 5, 1, 1, 1, 1, 5, 1, 1, 1, 1, 5, 1, 1, 1, 1, 5, 1, 1, 1, 1, 5, 1, 1, 1, 1] \]

> fibTree := proc(depth::integer)
>     # generates a non-balanced quadtree of given depth
>     local ones, zeros, k, ft, dp;
>     if depth = 0
>         then return [1]
>     else
>         if depth = 1
>         elif depth = 2
>         end if;
>         for k from 1 to 4 do
>             ft[k] := fibTree(dp[k]);
>         end do;
>         return [ 1+add( ft[k][1], k=1..4 ), seq( op(ft[k]), k=1..4 ) ];
>     end if;
> end proc:

> fibTree(3);
\[ [25, 13, 5, 1, 1, 1, 1, 5, 1, 1, 1, 1, 1, 1, 5, 1, 1, 1, 1, 5, 1, 1, 1, 1, 1, 1, 1] \]
Function \texttt{HilbertChilds} is equivalent to function \texttt{HilbertVL}, but generates an adaptive Hilbert order on the quadtree encoded by the list \texttt{vertices).

\begin{verbatim}
> HilbertChilds := proc(vertices::list)
    global SPACETREE, STPTR;
    STPTR := STPTR + 1;
    if SPACETREE[STPTR] = 1
        then
            mark(vertices);
            markcube(vertices[1], vertices[3]);
        else
            HilbertChilds( [ vertices[1],
                             mid2(vertices[1],vertices[4]),
                             mid4( op(vertices) ),
                             mid2(vertices[1],vertices[2]) ] );
            HilbertChilds( [ mid2(vertices[1],vertices[2]),
                             vertices[2],
                             mid2(vertices[2],vertices[3]),
                             mid4( op(vertices) ) ] );
            HilbertChilds( [ mid4( op(vertices) ),
                             mid2(vertices[2],vertices[3]),
                             vertices[3],
                             mid2(vertices[3],vertices[4]) ] );
            HilbertChilds( [ mid2(vertices[3],vertices[4]),
                             mid4( op(vertices) ),
                             mid2(vertices[1],vertices[4]),
                             vertices[4] ] );
        end;
    end proc;
\end{verbatim}

Function \texttt{HilbertQuadTree} calls \texttt{HilbertChilds} to generate the respective Hilbert order:

\begin{verbatim}
> HilbertQuadTree := proc(st::list)
    global points, cubes, SPACETREE, STPTR;
    local unitsquare;
    points := [];
    cubes := [];
    SPACETREE := st;
    STPTR := 0;
    unitsquare := [[0,0],[0,1],[1,1],[1,0]];
    HilbertChilds(unitsquare);
    return points;
end proc:

> HilbertQuadTree( fibTree(5)):
    dispcurv := plot( points,
                      scaling=CONSTRAINED, thickness=3, color=red, view=[0.1, 0.1]):
    dispqcube := plot( cubes,
\end{verbatim}
Partitioning

We apply the standard trick of cutting the generated list of Hilbert-iteration vertices into equal-sized sublists.

\[
\text{colours := [black, red, green, yellow, brown, magenta, cyan, navy, pink, grey, blue, khaki, coral];}
\]

\[\text{colours := [black, red, green, yellow, brown, magenta, cyan, navy, pink, grey, blue, khaki, coral]}\]
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 := HilbertQuadTree(fibTree(9)):parts := partition(pts, 11):
plot(parts, axes=BOXED, scaling=CONSTRAINED, thickness=3, color=colours);
Adaptive Hilbert Partition

Function `HilbertPartition` is equivalent to function `HilbertChilds`, but generates only one partition of an adaptive Hilbert order, given by the `first` and `last` index of the respective subsquares.

> `HilbertPartition := proc(vertices::list, first::posint,
last::posint)
    #option trace;
    global SPACETREE,STPTR;
    STPTR := STPTR + 1;
    if (SPACETREE[STPTR] = 1)
        then
            # leaf cell of the quadtree
            if (STPTR >= first) and (STPTR <= last)
                then
                    mark(vertices);
                    markcube(vertices[1], vertices[3]);
                end if;
            elif (STPTR + SPACETREE[STPTR] < first) or (STPTR > last)
                then
                    STPTR := STPTR + SPACETREE[STPTR] - 1; # subtree not in partition
                else
                    # expand subtree, if member of the desired partition
                    HilbertPartition( [ vertices[1],
                        mid2(vertices[1],vertices[4]),
                        mid4( op(vertices) ),
                        mid2(vertices[1],vertices[2]) ],
                        first, last );
                    HilbertPartition( [ mid2(vertices[1],vertices[2]),
                        vertices[2],
                        mid2(vertices[2],vertices[3]),
                        mid4( op(vertices) ) ], first, last );
                    HilbertPartition( [ mid4( op(vertices) ),
                        mid2(vertices[2],vertices[3]),
                        vertices[3],
                        mid2(vertices[3],vertices[4]) ],
                        first, last );
                    HilbertPartition( [ mid2(vertices[3],vertices[4]),
                        mid4( op(vertices) ),
                        mid2(vertices[1],vertices[4]),
                        vertices[4] ], first, last );
            end if;
        end proc;

Function HilbertQuadPart calls HilbertPartition to generate the respective Hilbert partition:
> HilbertQuadPart := proc(st::list, first::posint,
    last::posint)
global points, cubes, SPACETREE, STPTR;
local unitsquare;
points := [];
cubes := [];
SPACETREE := st;
STPTR := 0;
unitsquare := [[0,0],[0,1],[1,1],[1,0]];
HilbertPartition(unitsquare,first,last);
return points;
end proc:

> HilbertQuadPart( fibTree(5),38,73):
dispcurv := plot( points,
    scaling=CONSTRANDED, thickness=3, color=red, view=
    [0..1, 0..1]):
discube := plot( cubes,
    scaling=CONSTRANDED, thickness=1, color=black, view=
    [0..1, 0..1]):
display(dispcurv,dispcube);