UPPSALA
UNIVERSITET

Fractal Image Compression

Fredrik Lindén, Emil Södergren

Project in Computational Science: Report

December 2011

PROJECT REPORT

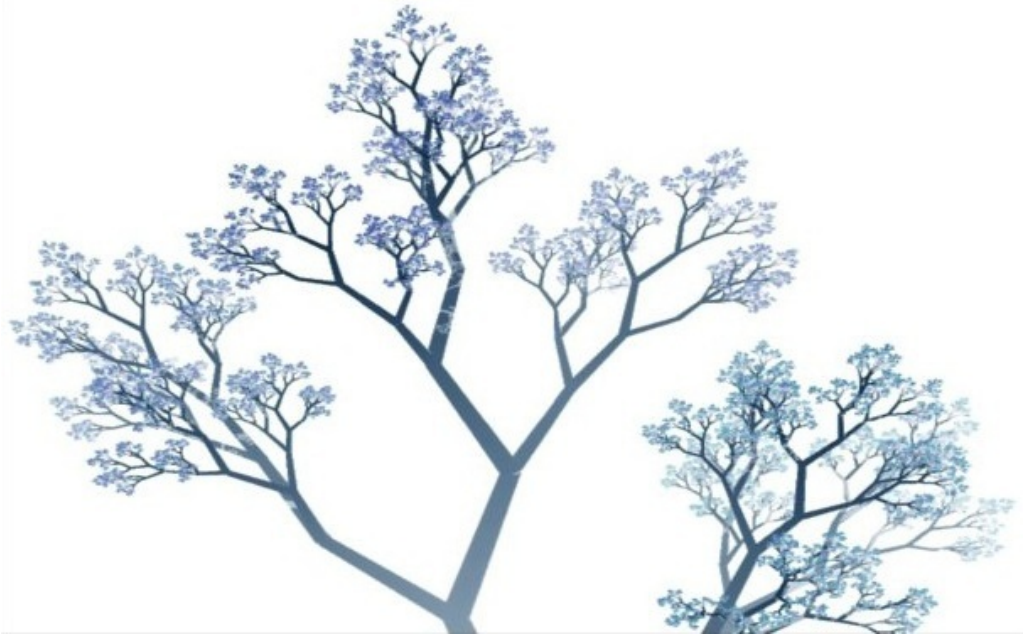
Institutionen för informationsteknologi

Innehåll

1 Introduction

In nature, many of the complex structures found can in a relatively simple way be described with a branch of mathematics called fractals. Fractals can be described as a self-repeating pattern that enlarges the structure by adding on smaller and smaller pieces of it self. One can think of this as a tree that shoots out small trees (branches) from the trunk which, in turn splits in to even smaller branches, building up the crown of the tree. The same pattern can be seen on cauliflowers where the big cauliflower head is built up by small parts that look exactly like the big head except, of course, the fact that they are smaller.

From a mathematical point of view the fractals are built up upon a few, very simple rules. The rules are then applied and reapplied to the set until the required level of refinement is reached.



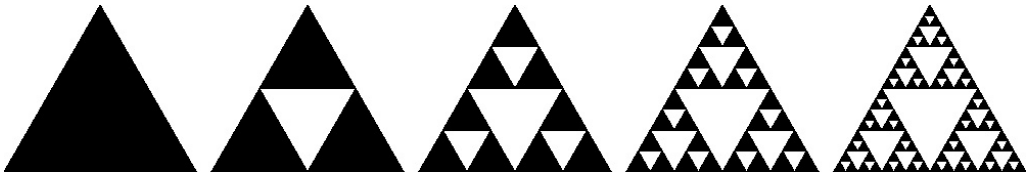
Figur 1.1: A tree shape created by using fractal rules, image from [2]

2 Theory

2.1 Fractals

To understand how fractals are built up it is useful to study one of the simplest fractals, the Sierpinski triangle. The method to create it can be formulated in different ways but in this report we use the shrinking method. We start with an equilateral triangle. In the first step we shrink the triangle to half its size in both height and width. Now we copy our small triangle three times and place these in our old triangle, letting each triangle

have two corners touching the others. The resulting figure will be of equal size as the first triangle but with a hole in the middle. Now we just repeat the shrinking and the multiplying for our new triangles. Figure 2.1 shows how the evolution of the Sierpinsky triangle looks for the four first steps.



Figur 2.1: The evolution of the Sierpinsky triangle, image from [3]

It is easy to see that this shrinking can be continued infinitely many times and if you zoom in on any smaller triangle, the pattern will be exactly the same. This is one of the fundamental properties of fractals, the fact that they mimic themselves on multiple levels.

Levels is an important concept and defines the amount of detail in an image. If we consider the rightmost triangle in Figure 2.1 as the original image, the four triangles to the left of it can be interpreted as the same image seen in higher levels. The level will then be some kind of detail filter.

2.2 Box counting

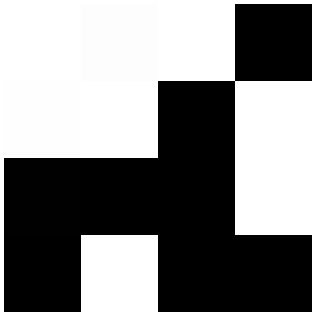
To classify fractals one needs some algorithms to extract interesting properties of them. Box counting is one of these algorithms. The idea is to count how many are the boxes that are filled in an image. If we consider an image similar to the one shown in Figure 2.2 we see that there are 8 pixels that are filled. In this case we see the black ones as filled. The algorithm can also be used in a higher level, meaning that we instead of looking at pixels, we can consider blocks of pixels. By choosing the mask as a block of 2-by-2 pixels one can see that this box count gives us 3 (the top left box is empty but the other three contain at least one filled pixel each). By choosing to view the image with a mask bigger than one pixel we can get a feeling for how scattered the pixels are in the image. In this case we know that the 8 pixels are appearing in only three of the four 2-by-2 blocks.

2.3 Lacunarity

The concept of lacunarity was first introduced by Benoît Mandelbrot in 1983. The word comes from the latin word *lacuna* which means *gap*. As the name suggest, the lacunarity measures how sparse an image is. Therefore, an image that is very dense gets a low lacunarity value while a sparse image get a high value. To calculate the lacunarity one uses the formula

$$\mathcal{L} = N \cdot \frac{\sum_{i=0}^{N} s_i^2 n_i}{\left(\sum_{i=0}^{N} s_i n_i\right)^2} \quad , \tag{2.1}$$

where s is the number of *states* that can occur with a specific mask. States can easiest be explained by looking at the image shown in Figure 2.2. If a mask that is the same



Figur 2.2: By counting the number of filled boxes in an image one can extract the box count. In this case, the black boxes are considered to be filled and thus, the boxcount is 8.

size as the image is used we get a vector $s = 0,1,2...,16$ which contains all the 17 possible states, 0 corresponding no pixels filled and 16 to all pixels filled. If we instead choose a smaller mask, e.g. 3-by-3 pixels, as shown in Figure 2.3 the vector s is a vector with 10 values $s = 0,1,2,...,9$. The index i specifies which element in s to use in the sum.

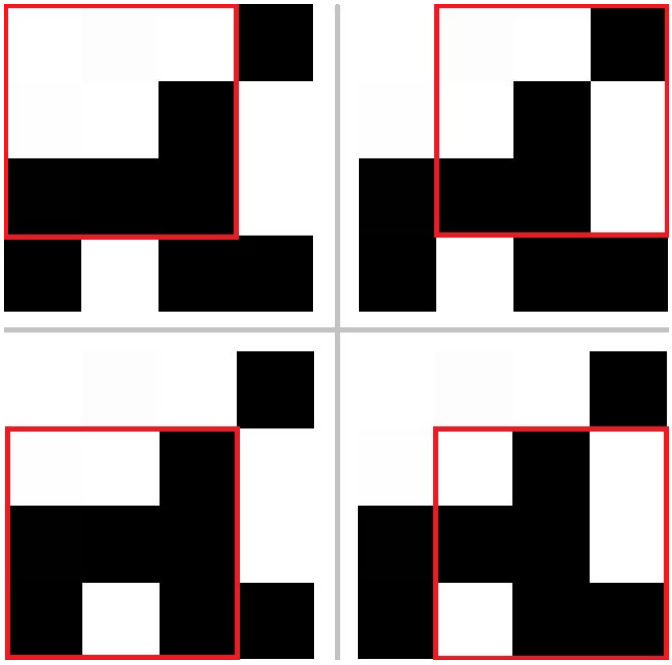
The variable N is the amount of different ways of placing the mask over the image. For the example above with a 4-by-4 image and a 4-by-4 mask $N = 1$. If we instead, as in Figure 2.3, choose a 3-by-3 mask, this will result in $N = 4$.

Finally, the variable n is a vector of the same length as s. The values in n shows how many times a certain state, s, appears. In Figure 2.3 one gets a vector n where the fourth element $n_4 = 2$ because in the two first images the mask contains four pixels. The fifth element $n_5 = 1$ because one of the masks contain 5 pixels and the sixth element $n_6 = 1$. All the other elements in n are zero. One can see that the sum of all elements in n is exactly N. This is always the case. [1]

By storing the different lacunarities for different levels and different mask sizes one achieves what we in this report refer to as a *lacunarity matrix*. We limit ourselves to masks that are an even power of 2. The lacunarity matrix, which contains the lacunarity value for different mask sizes on different image levels, is then a lower-triangular square matrix. The first row contains one value in the first column which is the lacunarity for the 2-by-2 level of the image with a 1-by-1 mask. Next row is for the 4-by-4 level of the image and in this row the first and second column contain the lacunarity counted with the 2-by-2 mask and the 1-by-1 mask, respectively.

3 Problem description

For this project the task was to investigate different methods of recreating a binary image showing subterranean cracks by using box counting and lacunarity. The easy part was to calculate these properties from the original image. After that we worked on the harder part of recreate the image using these values. For this we used different approaches with varying success.



Figur 2.3: Shows how the lacunarity is calculated, here with a mask size of 3 by 3 pixels.

4 Method of Compression

To calculate the lacunarity and box count we use the algorithms described above. We calculate both properties for all possible levels in the image. That means that a 32-by-32 pixel image obtain 5 values for box count and the lacunarity is a sub triangular 5-by 5-matrix. The function also returns the masks, corresponding to the different levels and the frame of the image which contains the coordinates for all the pixels set to 1 on the boarder of the image. The latter is only used by the Snake Algorithm, to be defined later. The code can be found in Appendix A.

5 Methods of Recreation

In this section we explain how our different recreation methods work. The three methods we have developed in the process of recreating the image are the ”Brute Force Algorithm”, the ”Semi-Randomized Algorithm” and the ”Snake Algorithm”.

The first one, as indicated by its name, is a very time and memory consuming method, where one inserts pixels, computes the lacunarity and compare it with the lacunarity of the original image. If the lacunarity differs, one discards the image and inserts new pixels and recomputes again.

The ”Semi-Randomized Algorithm” starts with an empty image and inserts random pixels one by one. After each insertion the lacunarity is computed and if the lacunarity improves, i.e. comes closer to the value of the original image, the pixel is stored as one.

This is repeated until the value for box count is fulfilled. If the lacunarity becomes worse, the pixel is set to zero.

Finally, the ”Snake Algorithm” starts from a set number of points (in our simulation we use the points where the cracks touch the image edge). From these points we let the cracks ”crawl” into the image like snakes. The cracks stop if they collide with each other or the image edge and new cracks are created if the number of set boxes is less than the original box count.

5.1 The Brute Force Algorithm

The box count gives us the right number of filled element in our image for each level, if we combine this knowledge, we can create smaller matrices with right number of filled element. Once we have done this, we place these smaller matrices in a random order and compute the lacunarity for the total image. If the lacunarity is not good enough we change places with the matrices. If the lacunarity is still not good enough, we rotate each matrix. This, however is a brute force method and becomes extremely computationally expensive Also, in a certain image level we can have several images with the same lacunarity and box count that look very different.

If a wrong image is acquired, this can not be discovered until one looks in the next level (or levels). This makes the method even more expensive because it is possible that one has to backtrack the solution to earlier levels. This can be helped by using a very extensive lacunarity matrix to store the lacunarity for all possible masks. This, however, will make the lacunarity matrix contain more data than it’s needed to store the image itself, which contradicts with the purpose of the method.

5.2 Semi-Randomized Algorithm

A slightly more promising approach is the Semi-Randomized Algorithm. The idea is to randomize pixels into the image at different levels. By doing this in a way that improves the lacunarity in each step, the idea is that the resulting image should be fairly similar to the original. The program randomizes a pixel somewhere in an empty image, calculates the lacunarity for that image and compares it with the original. If the lacunarity gets better than the last value, we keep the pixel. If not, it is set to zero. The image is recreated from the top level (4 boxes) and from there it expands the image down to the pixel level.

The code for this algorithm can be found in Appendix B.

5.3 The Snake Algorithm

This method resembles the semi-randomized algorithm. The difference is that we now utilize that cracks are continuous and when we randomize new points in the image we do it so they connect to the previous points. Also, we use the observation that cracks often are straight lines and therefore there is a larger probability that they proceed in a straight line. As the name of the algorithm suggests, we see the cracks as snakes crawling into the image. There is a certain probability in every step for the snakes to change direction. We believe that this probability can be connected with the lacunarity of the original image

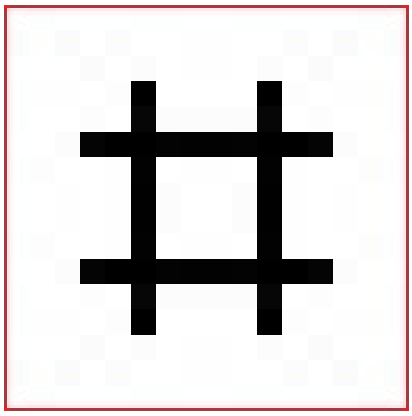
in some way. In the current implementation however, the chance of direction change is a fixed percent. The cracks grow until the number of pixels match the previously calculated box count or until they collide with each other or the image edge. If all cracks collide and the number of set pixels is less than the target box count, a new snake is started randomly from the tail of a random snake. Snakes, that are created and instantly collide, are neglected. The code for this algorithm can be found in Appendix C.

6 Results

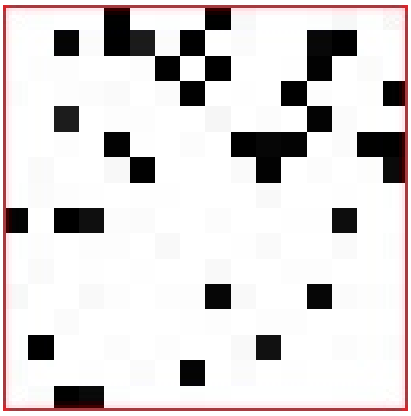
The first algorithm became too expensive to be sufficiently validated. Therefore, we do not have any runs of it. We conclude that the method is not very good nor interesting enough to spend time on.

The semi-randomized algorithm gives an image that appear to be very random and there is no visual resemblance with the original image. The recreated image in Figure 6.2 has the same box count and the lacunarity differs only in the second decimal to the original shown in Figure 6.1. Yet, the images are very very different. This suggests that the method is not very stable and will probably not produce any usable results with a reasonable amount of input data.

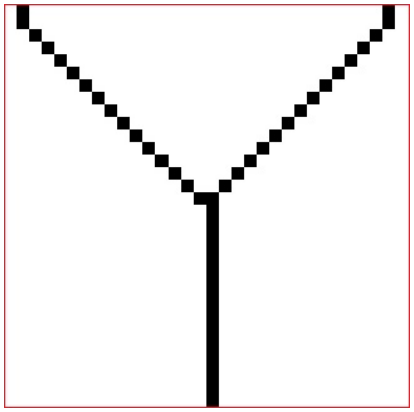
For our last algorithm the results are much more promising. When we ran the program for the image in Figure 6.3 we got the resulting image shown in Figure 6.4. The result looks really close to the original but the fact is that Figure 6.4 was the best of 20-30 runs. The best image is picked out by the program. It compares the lacunarity of the recreated images with the correct lacunarity values. The closest one is shown to the user. Since the snakes change direction with a certain, relatively low probability, many of the images looked more like the one in Figure 6.6.



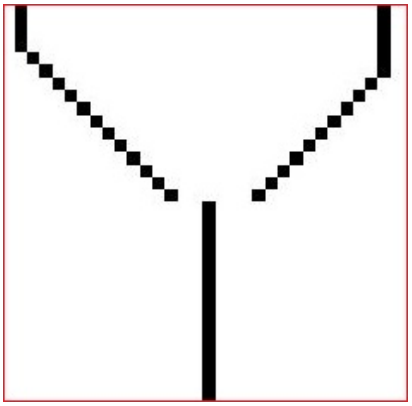
Figur 6.1: The original image.



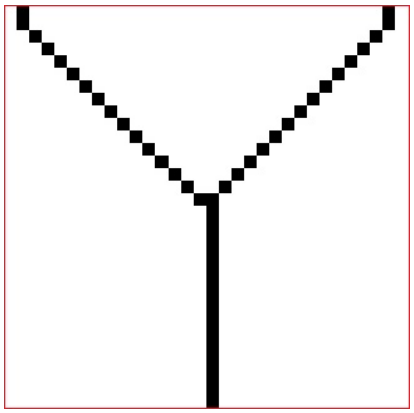
Figur 6.2: The recreated image.



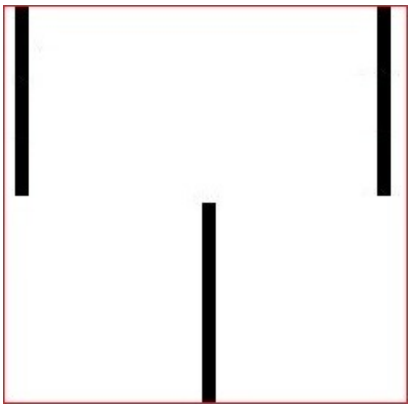
Figur 6.3: The original image.



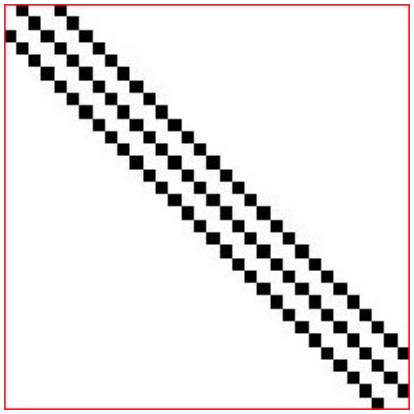
Figur 6.4: The recreated image.



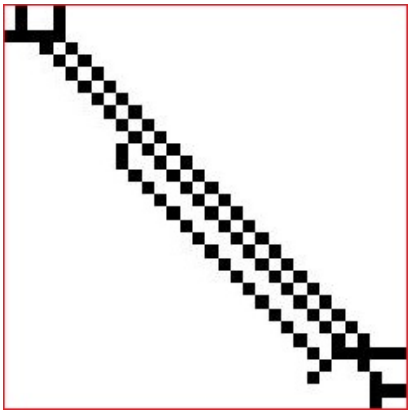
Figur 6.5: The original image.



Figur 6.6: The recreated image.

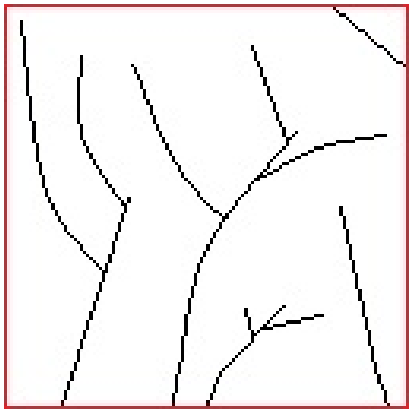


Figur 6.7: The original image.

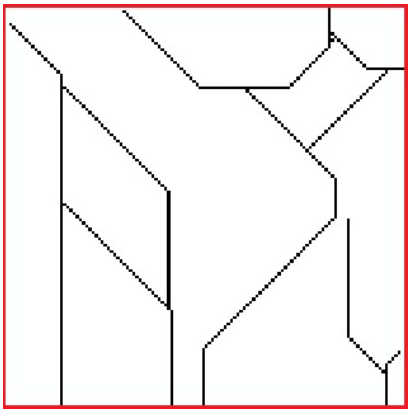


Figur 6.8: The recreated image.

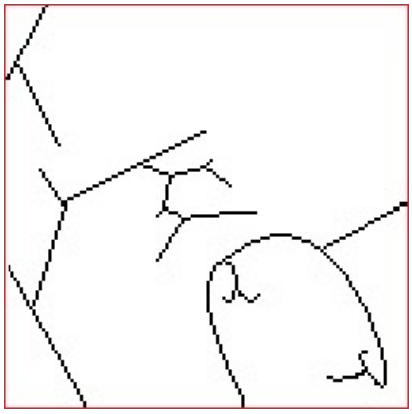
By just looking at the images in Figures 6.11 and 6.9, and their recreated counter parts,



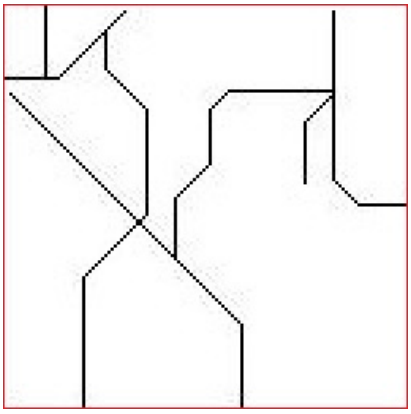
Figur 6.9: The original image.



Figur 6.10: The recreated image.



Figur 6.11: The original image.



Figur 6.12: The recreated image.

Tabell 1: The difference in lacunarity between the images in 6.11 and 6.12. Negative values correspond to a higher value in the recreated image. The rows in the table are the different levels going from 2-by-2 down to 128-by-128. The columns indicate the mask size going from 1-by-1 (leftmost column) to 64-by-64 (rightmost column).

0	0	0	0	0	0	0
0.1026	0.0199	0	0	0	0	0
0	0.1779	0.0331	0	0	0	0
0	0.1101	0.2429	0.0293	0	0	0
-0.2844	-0.0181	0.1633	0.2409	0.0147	0	0
-1.3190	-0.2506	-0.0338	0.1488	0.2231	0.0019	0
0	-0.4501	-0.1562	0.0186	0.2127	0.2409	0.0029

one does not see many similarities but as can be seen in Table 1 the difference in lacunarity is low in all levels. The box count is also very similar as one can see in Table 2. However, the original image is very complicated and hopefully cracks in the real world does not behave this way.

Tabell 2: Box count for the original image 6.11 and the recreated image 6.12 and also the difference between them.

level	original	recreated	difference
2x2	4	4	0
4x4	12	13	-1
8x8	32	32	0
16x16	72	72	0
32x32	150	144	6
64x64	279	256	23
128x128	471	471	0

Tabell 3: The compression for images of sizes 4 by 4 to 1024 by 1024. The second, third and fourth columns are given in bytes of data required for storage.

Image size	Full image	Best comp	Worst comp	Best %	Worst %
4x4	64	24	48	62.5	25.0
8x8	256	40	96	84.4	62.5
16x16	1024	60	148	94.1	85.6
32x32	4096	84	204	98.0	95.0
64x64	16384	112	264	99.3	98.4
128x128	65536	144	328	99.8	99.5
256x256	262144	180	396	99.9	99.9
512x512	1048576	220	468	¿99.9	¿99.9
1024x1024	4194304	264	544	¿99.9	¿99.9

The Figure 6.10 is recreated from image 6.9. It is the best produced image out of 2000 different simulations. Allthough the visual appearance is not very similar, the lacunarity and box count are.

To calculate the actual compression we use the formula (6.1). In this example we consider the original image and the compression data to be stored in equally sized format e.g. 4 byte floats. The variable N is the number of levels we use in the image i.e. for a 64-by-64 image there would be six levels, 128-by-128 gives seven levels, and so on. The parameter s is the number of starting points. It is multiplied by 2 to account for the fact that each point is a coordinate in a 2D matrix.

$$\frac{(2^N)^2}{\sum_{n=1}^{N} n + 2s - 1} \tag{6.1}$$

If we use this formula we get a compression shown in Table 3. The third column shows the best case scenario where there is only one starting point and the fourth column shows the worst case scenario where we store maximum amounts of starting points. This case can also be interpreted as a case where we store more points than just the starting points. Maybe in a future development one can store intermediate points to help the snakes find the right path.

7 Discussion

For the Semi-Randomized Algorithm we use a comparison with the lacunarity of the original picture for every inserted pixel. This comparison is done by calculating the sum of all the lacunarity values for all masks at the current level. This number is then compared with the previously calculated lacunarity for the original image. For the first steps, when the recreated image contains only a few pixels this method is not very good since every pixel makes an improvement to the overall lacunarity no matter where it is placed. This is a very unrefined way to do it and there are probably much better ways of determining whether the lacunarity improves with an added pixel in the early stages of recreation. But, even with this extension we feel like the method has more disadvantages than advantages.

For the snake algorithm there are several aspects that could be improved. As it is now, the only place a crack can start from is the edge of the image. This is of course easy to change but one have to take care so that one defines each crack only once. We have not made any attempt to define endpoints for the snakes but that would be a logical extension to the code. Also if there are cracks that join or split in the original image these points could be stored to help in the recreation process.

The algorithm extends the snakes cyclic which means that all snakes will have almost the same length in the recreated image. That means that short cracks will tend to be larger than the originals and vice versa. This could be helped by introducing endpoints or give the length of each crack as input to the recreation algorithm.

With a better knowledge about seismology and theoretical models of how the bedrock behaves, one can improve the behaviour of the evolving snakes. Especially, the part about directional changes and the likelihood of two cracks joining each other if they get close.

From the start we give our snakes a direction perpendicular to the edge it starts from. This condition can be problematic in some cases. As we can see in Figure 6.8, all snakes collide in the first few steps because they all start horizontally or vertically. This can be implemented differently to help out with the correctness of the recreated image. For the corners we have defined special cases and the snakes there starts with a 45 degree slope towards the center of the image. One could think of a solution where starting directions for the snake are determined by the distance to the closest edge. Perhaps by giving the snake a starting direction towards the center point of the image. This would give the snake a probability of starting at an arbitrary angle to its edge, rather than strictly perpendicular. It would of course be possible to store the starting angle, bypassing this problem completely. With the previous discussed extension of using endpoints this solution would be unnecessary.

As of now, the snakes have the largest probability to continue in the direction it headed in the last step. That means that if the snake turns it will be more probable for it to continue in the new direction, than to resume the old one. This is a major flaw with the implementation but is not a limit for the algorithm. We have discussed implementations where one use the start point to determine the main direction for the snake. This main direction would make the snake more attracted to continue on its path than to diverge away from it. We believe this to be a more accurate description of the behaviour for faults. Instead of comparing with the start point we have also discussed the possibility of using the snakes tail to determine the most likely direction. This would work in a similar

way but could maybe allow for more divergence in the path the snakes follow.

The snake algorithm is very well suited for parallel computation. One could let each processor handle one, or a group of snakes or let each processor take care of one area of the image. The first suggestion would imply a large overhead where the processors have to send and receive the tails of all snakes in each step so that they can determine if a collision occur. The second would take care of this problem but can instead be very load imbalanced so that some processors get areas where no snakes ever appear and others get lots of computations. This can of course be reduced by choosing a smarter computational grid. However, to create the snakes is not the most expensive part of the algorithm. When we ran a profiler it showed that nearly 60% of the total time was spent calculating the lacunarity matrix. This process is completely parallelizable and could therefore save a lot of time being done by a cluster of computers.

Referenser

[1] Yadvinder Malhi och Rosa María Román-Cuesta. "Analysis of lacunarity and scales of spatial homogeneity in IKONOS images of Amazonian tropical forest canopies". I: *Science Direct* 112 (jan. 2008): *Remote Sensing of Environment*, s. 2074–2087. URL: `http://www.geog.ox.ac.uk/~ymalhi/publications/publications2008/2008-remotesensing-lacunarity.pdf`.

[2] *Wikipedia page: Fraktal*. URL: `http://sv.wikipedia.org/wiki/Fraktal` (hämtad 2011-11-02).

[3] *Wikipedia page: Sierpinsky triangle*. Engelska. URL: `http://en.wikipedia.org/wiki/Sierpinski_triangle` (hämtad 2011-11-03).

A Compression algorithm

Here we list the code needed to run the compression part. The function `SLtransform` is the main function.

```
function [mask,boxDim,L,frameIndex]=SLtransform(fracImage,N,plotOn)
    close all

    if(N == 0)
        fracImage =im2bw(fracImage(1:length(fracImage),1:length(fracImage)));
    else
        fracImage =im2bw(fracImage(1:N,1:N));
    end

    drawImage(fracImage,plotOn);
    mask = getMaskFromImage(fracImage)
    boxDim = getBoxDimensionOfImage(fracImage,mask,plotOn)
    L = getTotalLacunarityOfImage(fracImage,mask,plotOn)
    frameIndex = getFrame(fracImage)
end

function b = drawImage(fracImage,plott)
    if (( plott == 1 ) || strcmpi(plott,'yes') ||...
          strcmpi(plott,'y') || strcmpi(plott,'plot'))
        imshow(fracImage)
    end
end

%
% MASK = GETMASKFROMIMAGE(FRACIMAGE)
%   Returns a vector containing the maximum number of masks that can be
%   retrieved from the image FRACIMAGE. It needs an image that is a square
%   with sides 2^n where n is integer. The mask is set to go from top
%   level (4 equally sized squares) to bottom level (1 pixel). We
%   neglect the case where length(MASK)==length(FRACIMAGE) for obvious
%   reasons.
%
function mask = getMaskFromImage( fracImage )
    mask = zeros( log2(length(fracImage)), 1 );
    for i = 1:length(mask)
        mask(i) = 2^(i-1);
    end
    mask = mask(end:-1:1); % Flip the mask vector
end

%
%   BOXDIMENSION returns a polyfit of the box dimension of the image.
%     BOXDIMENSION = GETBOXDIMENSIONOFIMAGE(FRACIMAGE,PLOT)
%     BOXDIMENSION is a 2x1 vector containing the K and M value of the formula
%     Y = K*X+M which is a polyfit of the boxcount for different levels in
%     FRACIMAGE. The levels goes from 2^2n (n is integer) squares down to
%     2x2 pixel boxes for the image FRACIMAGE. The resulting box counting
%     vector, containing the number of set pixels at each level, is plotted
%     versus the vector containing the levels. The BOXDIMENSION is then the
%     slope of the graph which is obtained by performing polyfit on the
%     graph mentioned.
%
function boxDimension = getBoxDimensionOfImage(fracImage,maskLengths,plott)
    levels = length(maskLengths);  % get number of levels from image
    nrOfSetPixels = zeros( levels, 1 );
    for i = 1:levels
        nrOfSetPixels(i) = countSetPixelsInImage(fracImage,maskLengths(i));
    end
    dimFit = polyfit(log(maskLengths),log(nrOfSetPixels),1);
    boxDimension = nrOfSetPixels;

    %%%
    %%% The following section draws the graph and the 1-degree polyfit of
    %%% the values if set by user.
    %%%
```

```
    if (( plott == 1 ) || strcmpi(plott,'yes') ||...
          strcmpi(plott,'y') || strcmpi(plott,'plot'))
        figure
        plot(log(maskLengths),log(nrOfSetPixels),'b');
        grid on
        hold on
        plot(log(maskLengths),dimFit(1)*log(maskLengths)+dimFit(2),'r')
        hold off
        xlabel('log(maskLengths)')
        ylabel('log(nr of set pixels)')
    end
end

%
% COUNTSETPIXELSINIMAGE counts the non-zero pixels in an image.
%     [SETPIXELCOUNT, TOTALNUMBEROFBLOCKS] =
%                                     COUNTSETPIXELSINIMAGE(FRACIMAGE,MASKLENGTH)
%     returns the number of set (non-zero) pixels in an image fracImage.
%     The function takes the image, FRACIMAGE, and length of the mask,
%     MASKLENGTH.
%
%     The MASKLENGTH partitions the image into blocks of MASKLENGTH^2
%     elements.
%
%     For example, if MASKLENGTH is set to 1 the function counts all set
%     pixels in the image. If the mask is set to 2 the image is
%     partitioned in to 2 by 2 blocks and the function counts all blocks
%     which have at least one set pixel in them. The total number of
%     blocks created by COUNTSETPIXELSINIMAGE is returned in
%     TOTALNUMBEROFBLOCKS. That is, if MASKLENGTH = length(fracImage) the
%     function returns [1,1] if the image contains at least one non-zero
%     pixel. [0,1] otherwise.
%
%     Warning messages result if IMAGELENGTH or MASKLENGTH is not of the
%     form 2^n where n is an integer. Also if IMAGELENGTH < MASKLENGTH < 1.
%
function [setPixelCount,totalNrOfBlocks] = countSetPixelsInImage(fracImage,maskLength)
imageLength=length(fracImage);
nrOfMasks = imageLength/maskLength;
setPixelCount = 0;
totalNrOfBlocks = nrOfMasks^2;
for rmi = 0:nrOfMasks-1     % rowMaskIndex loops over all masks
    for cmi = 0:nrOfMasks-1 % colMaskIndex
        ml = maskLength;    % shorthand for maskLength to be used as index in fracImage
        pixelIsSet = isAnyPixelSetInMask(fracImage(rmi*ml+1:(rmi+1)*ml,...
        cmi*ml+1:(cmi+1)*ml)); % Sending the current part of fracImage to the function
        if ( pixelIsSet )
            setPixelCount = setPixelCount + 1;
        end
    end
end
end

function L=getTotalLacunarityOfImage(fracImage,mask,plott)
    L=zeros(length(mask));
    for i=1:length(mask)
        thisImage = getSubImage(fracImage,mask(i));
        for j=length(mask):-1:length(mask)-i+1
            L(i,length(mask)-j+1)= lac2(thisImage,mask(j));
        end
    end

    if (( plott == 1 ) || strcmpi(plott,'yes') ||...
          strcmpi(plott,'y') || strcmpi(plott,'plot'))
        figure
        epsilon=mask./length(fracImage);
        plot(log(epsilon),log(L),'b')
        grid on
        xlabel('log(maskLength)')
        ylabel('log(Lacunarity)')
    end

end
```

```
function outMat = getSubImage(inMat, mask)
nrOfMasks = length(inMat)/mask;
outMat = zeros(nrOfMasks );

for rmi = 0:nrOfMasks-1     % rowMaskIndex loops over all masks
    for cmi = 0:nrOfMasks-1 % colMaskIndex
        m = mask;           % shorthand for maskLength to be used as index in fracImage
        pixelIsSet = isAnyPixelSetInMask(inMat(rmi*m+1:(rmi+1)*m,...
        cmi*m+1:(cmi+1)*m)); % Sending the current part of fracImage to the function
        if ( pixelIsSet )
            outMat(rmi+1,cmi+1) = 1;
        end
    end
end

function L = lac2(fracImage,mask)
    imageLength=length(fracImage);
    totalBox=(imageLength-mask+1)^2;%N
    n=zeros(mask^2+1,1);

    for i=1:imageLength%yled
        for j=1:imageLength%xled
            if(fracImage(i,j)==2)
                fracImage(i,j)=1; %säter 2or till 1or
            end
        end
    end

    for col=1:(imageLength-mask+1)
        for row=1:(imageLength-mask+1)
            temp=sum(sum(fracImage(row:row+mask-1,col:col+mask-1)));
            n(temp+1)=n(temp+1)+1;
        end
    end
    n=n./totalBox;%Q
    s=find(n)-1;
    z1 = zeros(length(s),1);%
    z2 = zeros(length(s),1);%
    for i=1:length(s)
        z1(i)=s(i)*n(s(i)+1);
        z2(i)=s(i)^2*n(s(i)+1);
    end
    L=sum(z2)/(sum(z1))^2;
end

function index=getFrame(fracImage)
    w=zeros(length(fracImage));

    w(1,:)=fracImage(1,:);                  %first row
    w(2:end-1,1)=fracImage(2:end-1,1);      %first col,
    w(2:end-1,end)=fracImage(2:end-1,end);  %last col
    w(end,:)=fracImage(end,:);              %last row

    [index(:,1),index(:,2)]=find(w);        %return startpoints as [x y]
end
```

B Recreation with Semi-Randomized Algorithm

```
function A = recreateImage(BC,Lac,mask)

imageSize = 2*mask(1);
A = 1;
lastTempLac = zeros(1,length(BC));
tempLac = zeros(1,length(BC));
nrOfSubBoxes = getLevelsFor(imageSize).^2;
BC = nrOfSubBoxes-BC;
```

```
for level = 1:length(BC)
    disp(level);
    % Stores all filled pixels as 1
    A = expandMatrix( A );
    % number of set boxes on this level < nrOfSubBoxes
    if ( BC(level) ~= 0 )
        [indx indy] = chooseRandomUnSetPixel(A);
        % randoms an index for empty element in A
        while (isempty(indx) == 0 && (length(find(A==0))<BC(level)))
            % Set the randomly chosen pixel to 0
            A(indx,indy) = 0;
            for i = 1:level
                % calculate lacunarity for the testmatrix
                tempLac(i) = lac(A,2^(i-1));
            end
            if (abs(sum(tempLac - Lac(level,:))) < ...
                                    abs(sum(lastTempLac - Lac(level,:))))
                % keep if lacunarity gets better, otherwise...
                lastTempLac = tempLac;
            else
                % ...discard and "one" the suggested pixel
                A(indx,indy) = 1;
            end
            % randoms an index for empty element in A
            [indx indy] = chooseRandomUnSetPixel( A );
        end
    end
end

imshow(A,'border','tight');
end


function levels = getLevelsFor( imageLength )
A = zeros(imageLength);
levels = getMaskFromImage(A);
levels = levels(end-1:-1:1);
levels = [levels;imageLength];
return;


function B = expandMatrix(A)
B=zeros(length(A)*2);
    for col=1:length(A)
        for row=1:length(A)
            if(A(row,col)==1)
                xcord=2*row-1:2*row;
                ycord=2*col-1:2*col;
                B(xcord,ycord)=1;
            end
        end
    end
end


function [indx indy] = chooseRandomUnSetPixel( inMat )

[xcoord, ycoord] = find( inMat == 1 );
randomIndex = randi(length(xcoord),1,1); % Random one of the elements
indx = xcoord(randomIndex);              % Return that element
indy = ycoord(randomIndex);
end
```

C Recreation with Snake Algorithm

```
clear all


 [mask,boxDim,Lac,index] = SLtransform(imread('../images/tes.tif'),0,'n');
 %load('fredrikfrak.mat');
img = FracImage(boxDim,Lac,index,mask(1)*2);
class(img)
```

```
boxDim = boxDim

N=4; %runs

H=cell(N,1);
K=zeros(N,1);
difMat=cell(N,1);

%probVec= 1:-1/log2(img.totalSize):1/log2(img.totalSize)
figure

for rep=1:N
    rep
    A=evolveSnakes(img,log2(img.totalSize));
    H{rep}=A;
    L2 = getTotalLacunarityOfImage(A,mask,'n');
    b = getBoxDimensionOfImage(A,mask,'n');
    difMat{rep}=(Lac-L2);
    K(rep)=sum(sum(abs(difMat{rep})));
end
K

in=find(K==min(K))
imshow(H{in(1)})


classdef FracImage
    properties
        boxDim
        lac
        frameIndex
        snakes
        currentBox
        totalSize
    end
    methods
        function obj = FracImage(boxDim,lac,frameIndex,totalSize)
            obj.totalSize=totalSize;
            obj.boxDim=boxDim;
            obj.lac = lac;
            obj.frameIndex = frameIndex;
            obj.currentBox = 0;
            obj.snakes = {0};
        end

        function frame = levelFrameIndex(obj,level)
            % Converts the frame for diffrent levels
            frame = ceil(obj.frameIndex.*(2^level)/obj.totalSize);
        end

        function A = evolveSnakes(obj,level)
            frame = levelFrameIndex(obj,level); % contains set pixels on the frame for this level
            obj.snakes = cell( size( frame,1 ),1 );
            totalNrOfSnakes=0;
            for i = 1:size(frame,1)
                % Here we create the snakes that are starting from the edge
                obj.snakes{i} = Snake(frame(i,:),2^level);
                totalNrOfSnakes=1+totalNrOfSnakes;
            end

            snakeIndex = 1;
            vikt = sum(1:(level))*1.5;% The probability weight

            while( anySnakeIsAlive(obj) )
                % Grow all snakes until they are dead
                obj = killer(obj,snakeIndex,level);
                [obj.snakes{snakeIndex}, obj.currentBox] = move(obj.snakes{snakeIndex}, obj.currentBox,vikt);
                obj = collision(obj,snakeIndex);
                snakeIndex = mod(snakeIndex,size(obj.snakes,1))+1;% Get next snake from cell array

                if( ~anySnakeIsAlive(obj) && obj.currentBox < obj.boxDim(level))
                    % If all snakes are dead but the total number of pixels
                    % set are too few
                    if (  size(getTail(obj.snakes{end}),1) < 2 )
```

```
                    % If last snake is shorter than 2 pixels
                    % Ignore it!
                    totalNrOfSnakes=totalNrOfSnakes-1;
                end
                obj=splitSnake(obj,level,totalNrOfSnakes);
                totalNrOfSnakes=totalNrOfSnakes+1;
            end
        end
        A=snakeImage(obj);
    end

    function obj = splitSnake(obj,level,totalNrOfSnakes)
        % Splits the snake into two different snakes
        start = [0 0];
        while( size(start,1) < 4 )
            snakeNr = randi(totalNrOfSnakes,1,1);
            start = getTail(obj.snakes{snakeNr});
        end
        tailNr = randi([3 size(start,1)-1],1,1);
        obj.currentBox=obj.currentBox-1;
        obj.snakes{totalNrOfSnakes+1,1} = Snake(start(tailNr,:),2^level);
    end

    function obj = killer(obj,snakeIndex,level)
        % If the nr of set pixels are equal to the allowed nr pixels we kill all snakes
        if(obj.currentBox>=obj.boxDim(level))
            obj.snakes{snakeIndex} = killSnake(obj.snakes{snakeIndex});
        end

    end
    function obj = collision(obj,snakeIndex)
        % Kills the snake when it crashes into another snake
        thispos = getPos(obj.snakes{snakeIndex});
        for i=1:size(obj.snakes,1)
            A=getTail(obj.snakes{i});
            for j=1:size(A,1)
                if(thispos(1) == A(j,1) && thispos(2) == A(j,2))
                    obj.snakes{snakeIndex}=killSnake(obj.snakes{snakeIndex});

                end
            end
        end
    end

    function A = printSnakes(obj,frame,t)
        disp('All snakes')
        if(t==0)
            for i = 1:size(frame,1)
                disp(obj.snakes{i})
            end
        else
            disp(obj.snakes{t})
        end
    end
    function A = anySnakeIsAlive(obj)
        A = 0;
        for i = 1:size(obj.snakes,1)
            A = A + isAlive(obj.snakes{i});
        end
    end
    function A = snakeImage(obj)
        % Creates the final image with all snakes
        A = zeros(obj.totalSize);
        for i = 1:size(obj.snakes,1)
            tail = getTail(obj.snakes{i});
            for j = 1:size(tail,1)
                A(tail(j,1),tail(j,2)) = 1;
            end
        end
    end
end
end
```

```
classdef Snake
    properties
        pos           %current position
        tail          %previous position
        direction           %direction
        lives         %life
        totalSize     %Total size of world
    end
    methods
        function obj = Snake(start,totalSize)
            obj.pos = start;
            obj.tail = [];
            obj.lives = 1;
            obj.direction = [0,0];
            obj.totalSize = totalSize;
        end

        function [obj,currentBox] = move(obj,currentBox,vikt)
            %  Returns the new direction, position and if its alive a pixel is set

            if( isAlive(obj) )
                % If NOT on boundary
                if(obj.pos(1)>2 && obj.totalSize-1 > obj.pos(1) && obj.pos(2)>2 && obj.totalSize-1 > obj.pos(2))
                    p = possibleDir(obj);
                    obj.direction = newDir(obj,p,vikt);
                else
                    obj = boundaryCheck(obj,vikt);
                    if(~isAlive(obj))
                        return
                    end
                end

                obj.tail = [obj.tail;obj.pos]; %Add position to tail
                obj.pos = obj.pos + obj.direction; % Update position
                currentBox = currentBox+1;
            end
        end

        function lives = isAlive(obj)
            % Check if the snake is alive
            lives = obj.lives;
        end

        function obj = killSnake(obj)
            % Kills the snake
            obj.lives = 0;
        end

        %stopa in detta i new direction
        function sto = getPreviosDir(obj)
            i=1;%får inte vara udda tal i sådant falll ändra h=totsize
            if(length(obj.tail)>i)
                direction= obj.pos-obj.tail(i,:);
                if(direction(1)==0 && direction(2)~=0)
                    hypotunusan = sqrt(sum((obj.pos-obj.tail(i,:)).^2));%kord
                    Z = obj.pos(2)-obj.tail(1,2);%
                    Q = obj.pos(1)-obj.tail(1,1);%
                    vinkel = abs(atan(Q/Z));%
                    if(direction(2)>0)
                        Y = hypotunusan*sin(pi/4-vinkel);
                        X = hypotunusan*sin(pi/4+vinkel);
                    else
                        X = hypotunusan*sin(pi/4-vinkel);
                        Y = hypotunusan*sin(pi/4+vinkel);
                    end

                elseif(direction(1)~=0 && direction(2)==0)
                    hypotunusan = sqrt(sum((obj.pos-obj.tail(i,:)).^2));%kord
                    Z = obj.pos(1)-obj.tail(1,1);%
                    Q = obj.pos(2)-obj.tail(1,2);%
                    vinkel = abs(atan(Q/Z));%
                    if(direction(1)>0)
                        X = hypotunusan*sin(pi/4-vinkel);
```

```
                Y = hypotunusan*sin(pi/4+vinkel);
            else
                Y = hypotunusan*sin(pi/4-vinkel);
                X = hypotunusan*sin(pi/4+vinkel);
            end
        else
            skift = sqrt(sum((obj.pos-obj.tail(i,:)).^2));%kord
            if(direction(1)>0 && direction(2)>0 || direction(1)<0 && direction(2)<0)
                X = obj.pos(1)-obj.tail(1,1);%
                Y = obj.pos(2)-obj.tail(1,2);%
            elseif(direction(1)<0 && direction(2)>0 || direction(1)>0 && direction(2)<0)
                Y = obj.pos(1)-obj.tail(1,1);%
                X = obj.pos(2)-obj.tail(1,2);%
            end
            vinkel = abs(atan(Y/X));%
            Z = skift*cos(pi/4-vinkel);%
        end
    else
        h=obj.totalSize+1-i;
        pos = obj.pos;
        if(pos(1)<=h/2 && pos(2)<=h/2)
            X = (h/2-1)+pos(1);
            Y = (h/2-1)+pos(2);
        elseif(pos(1)>=h/2 && pos(2)>=h/2)
            X = (h/2)-pos(1);
            Y = (h/2)-pos(2);
        elseif(pos(1)>=h/2 && pos(2)<=h/2)
            X = -(h/2)+pos(1);
            Y = (h/2-1)+pos(2);
        elseif(pos(1)<=h/2 && pos(2)>=h/2)
            X = -(h/2)+pos(2);
            Y = (h/2-1)+pos(1);
        end

        skift = sqrt(sum(X^2+Y^2));%kord
        vinkel = abs(atan(Y/X));%
        Z = skift*cos(pi/4-vinkel);%
        prob = 1/(abs(Y)+abs(X)+abs(Z));%
        sto = abs([X,Z,Y])*prob;%
    end

    prob = 1/(abs(Y)+abs(X)+abs(Z));%
    sto = abs([X,Z,Y])*prob; %X sanolikheten att svänga vänstr Y åt höger och Z rakt fram
end

function direction = newDir(obj,vecDir,vikt)
    sto=getPreviosDir(obj);
    styrning=vikt;
    r=rand*styrning;
    if(r<sto(1))
        direction = vecDir(1,:);
    elseif(r>=sto(1) && r<sto(3)+sto(1))
        direction = vecDir(3,:);
    elseif(r>=sto(3)+sto(1) && r<styrning)
        direction = vecDir(2,:);
    end
end


function vecDir = possibleDir(obj)
    %   Returns the possible directions a snake can take based on how
    %   it moved in the last step
    direction = obj.direction;
    %       ->  [-1,-1] ->[-1, 0] ->[-1, 1]
    %                    ^                V
    % A =       [ 0,-1]    [ 0, 0]    [ 0, 1]
    %                    ^                V
    %           [ 1,-1] <-[ 1, 0] <-[ 1,1]
    % The cell array A holds all the 8 directions a snake can take
    % orderd as above
    A={[-1,-1],[-1,0],[-1,1],[0,1],[1,1],[1,0],[1,-1],[0,-1]};
    if(direction==[0,0])
        % Move to a random direction
```

```
        midpoint = [obj.totalSize/2,obj.totalSize/2];
        if(obj.pos == midpoint)
            direction=A{randi(length(A),1,1)};
        elseif(obj.pos(1)>3 || obj.pos(1)<obj.totalSize-3 ||...
                obj.pos(2)>3 || obj.pos(2)<obj.totalSize-3 )
            direction = midpoint-obj.pos;
        else
            direction=A{randi(length(A),1,1)};
        end
        direction=round(direction/sqrt(sum(direction.^2)));

    end
    % Chooses the direction
    for i=1:8
        if(direction==A{i}) % Find the last direction
            if(i == 1 )
                vecDir = [A{8};A{1};A{2}];
            elseif(i==8)
                vecDir = [A{7};A{8};A{1}];
            else
                vecDir = [A{i-1};A{i};A{i+1}];
            end
        end
    end
end

function obj = boundaryCheck(obj,vikt)
    %   Returns the new direction or,
    %   if the snake crashes into a wall / edge: KILL IT!

    A=.8; %probability to change direction at the edges
    xl=2;
    yl=obj.totalSize-1;
    % all snakes larger than two pixels
    if (size(obj.tail,1)>2)
        % upper or lower edge
        if(obj.pos(1) == xl || obj.pos(1) == yl) %
            % Snake runs straight into wall
            if(abs(obj.direction(1))==1 && obj.direction(2)==0)
                obj = killSnake(obj);
                % If snake runs diagonal into wall
            elseif(sum(abs(obj.direction))==2)
                %obj.direction = obj.direction.*[0,1];%[-1,1]
                obj = killSnake(obj);
                % The snake i moving along with the edge
            elseif(obj.pos(1) == xl && obj.direction(1)==0 && obj.direction(2)==1)
                if(rand>=A)
                    obj.direction = [0,1];
                else
                    obj.direction = [1,1];
                end
                % The snake i moving along with the edge
            elseif(obj.pos(1) == xl &&obj.direction(1)==0 && obj.direction(2)==-1)
                if(rand>=A)
                    obj.direction = [0,-1];
                else
                    obj.direction = [1,-1];
                end
                % The snake i moving along with the edge
            elseif(obj.pos(1) == yl && obj.direction(1)==0 && obj.direction(2)==1)
                if(rand>=A)
                    obj.direction = [-1,1];
                else
                    obj.direction = [0,1];
                end
                % The snake i moving along with the edge
            elseif(obj.pos(1) == yl && obj.direction(1)==0 && obj.direction(2)==-1)
                if(rand>=A)
                    obj.direction = [-1,-1];
                else
                    obj.direction = [0,-1];
                end
            end
```

```
                % left or right edge
            elseif(obj.pos(2) == xl || obj.pos(2) == yl)
                % Snake runs straight into wall
                if(obj.direction == [0,1] )
                    obj = killSnake(obj);
                    % Snake runs straight into wall
                elseif(obj.direction == [0,-1])
                    obj = killSnake(obj);
                    % If snake runs diagonal into wall
                elseif(sum(abs(obj.direction))==2)
                    obj = killSnake(obj);%obj.direction = obj.direction.*[1,0];%[1,-1];
                    % The snake i moving along with the edge
                elseif(obj.pos(2) == xl && obj.direction(1)==-1 && obj.direction(2)==0)
                    if(rand>=A)
                        obj.direction = [-1,1];
                    else
                        obj.direction = [-1,0];
                    end
                    % The snake is moving along with the edge
                elseif(obj.pos(2) == xl && obj.direction(1)==1 && obj.direction(2)==0)
                    if(rand>=A)
                        obj.direction = [1,1];
                    else
                        obj.direction = [1,0];
                    end
                    % The snake is moving along with the edge
                elseif(obj.pos(2) == yl && obj.direction(1)==1 && obj.direction(2)==0)
                    if(rand>=A)
                        obj.direction = [1,-1];
                    else
                        obj.direction = [1,0];
                    end
                    % The snake is moving along with the edge
                elseif(obj.pos(2) == yl && obj.direction(1)==-1 && obj.direction(2)==0)
                    if(rand>=A)
                        obj.direction = [-1,-1];
                    else
                        obj.direction = [-1,0];
                    end
                end
            end
        else
            % Move towards the center of the image
            p = possibleDir(obj);
            obj.direction = newDir(obj,p,vikt);
        end
    end

    function direction = getDir(obj)
        direction = obj.direction;
    end

    function pos = getPos(obj)
        pos = obj.pos;
    end

    function tail = getTail(obj)
        tail = obj.tail;
    end

    function A = printAll(obj)
        disp(obj)
    end
end
end
```