Xelular Sets Code Documentation

Badal Yadav and Tim Poston
Forus Health
Bangalore 560082
India

March 31, 2014

Contents

1 Introduction: xelular sets

The mathematics of shapes — geometry — began with shapes built from straight lines (all those triangles and n-gons in school geometry), and then very specific curves like circles (at least as old as the wheel) and conics (ellipses, *etc.*), which were first(?) studied by Menaechmus, a tutor to Alexander the Great.

Much later came general parametrized curves and surfaces $c(s) = (x(s), y(s))$ or $v(s) = (x(s), y(s), z(s))$, where $x(s)$ and its friends could have any formula: $\sin(2s)$, or $s^3 - \sqrt{s}$, or anything you could define.

But these were not enough to describe all the shapes we find in the world.

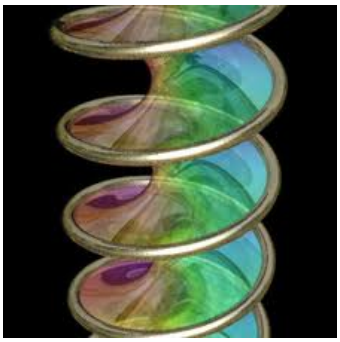
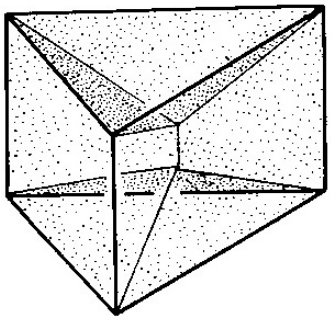
(a)  (b)  (c) 

Figure 1: A soap film surface parametrisable by radius and height; the branchings usual in real foams; and a fractal shape.

In the 19th Century, set theory allowed a shape to be specified as an arbitrary set of points. It took a while for mathematicians to exploit this properly: for instance, they stuck with smooth shapes (Fig. 1a) for a century before addressing what soap films actually do. It was even later that they applied fractals to the real world.

Figure 2: Satellite view of part of the Amazon river system; lightning; blood vessels in the eye.

Fractals are complicated on an infinite range of scales, so it takes remarkable cunning to handle them well on computers, but the awkwardness of forms like (Fig. 1b) is mainly due to the mental habit of parametrisation. They can be parametrized along the curves between branch points, but any change in their branching structure (topology) changes the collection of such curves. It changes which curves meet which, and often their number.

Soap films begin as blobs of soapy water, and shrink down to smooth surfaces, whose topology are emerging properties. By analogy, we work numerically with sets of points that need not always be curves, and equip them with a dynamic, so that they model not just static shapes, but shapes that change and respond to their surroundings. The early motivation (which remains a goal) was to model growth and form for structures that change shape radically as they develop, such as trees, leaves and brains, but current work is focussed on an analogy to surface tension. Depending on the details, such a dynamic can push a set to become 'thread-like' in 2D or 3D, or surface-like in 3D — but allows branching. It can be combined with forces derived from an image, as in computational 'snakes' (http://en.wikipedia.org/wiki/Active_contour), but without the topological preconceptions built into that class of model that ill fit it to adapt to branched edges or curves in the image. This branched-curve fitting is the use implemented in the C++ code described here.

To avoid dealing with arbitrarily large numbers of points, and avoid the local complexity of fractals, we divide the plane into hexagonal tiles (Fig. 3), which for present purposes we call **xels**. A set with at most one point per xel is called a **xelular set**, or a **xelset**. Note that there is no relation (at this level) with pixels. Numerically, a point is represented by a pair (x, y) of floating point numbers. This allows greater precision — and smoother curves — than with a set limited to pairs of pixel grid points (`i,j`). We refer to two points as **neighbours** if the hexagonal xels they occupy share a side. (Note that unlike square tiles, xels touch along a side or not at all: They cannot meet at a corner only. This greatly simplifies image topology.)

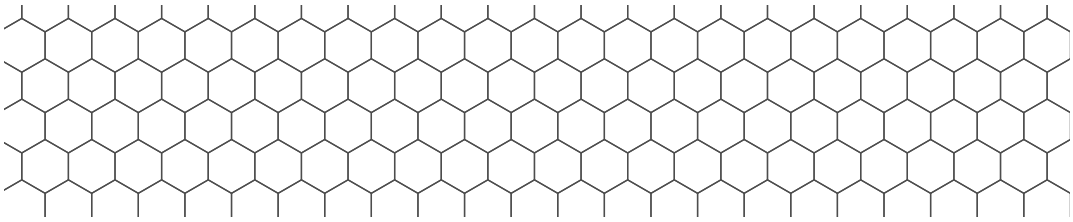


Figure 3: Subdivision of 2D space by a hexagonal tiling. The code takes each hexagon to have two vertical sides, none horizontal.

In a hexagonal array the vertical spacing between rows is only $\sqrt{3/4}$ the spacing of points along a row, so that some geometric numbers must be set (Fig. 4), both for display and for decisions such as which xel a point with coordinates (x, y) belongs to.

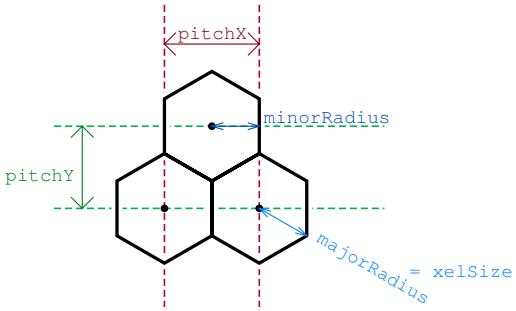


Figure 4: Four numbers computed in `param2d.cpp` from the one parameter `xelSize` (the side length of the hexagons in pixel units).

There are many possible implementations of xelset kinetics and dynamics in code. One is to associate memory with each xel, in which we store presence/absence of a point in it, and the floating coordinates `p` $= (x, y)$ of that point if present. Each xel updates using data from its immediate neighbours, and perhaps transfers a point to or from a neighbour. On a serial computer it is often faster to iterate over the points of the xelset: this avoids any computation for empty xels that remain empty, which can vastly outnumber the occupied ones. In the implementation described here, motivated by the availability of GPU parallelism, we iterate over xels: the relevant function is `cpuRun()`, defined in the file `xel2d.cpp`. Each xel also stores, as a constant, the floating (x, y)-coordinate pair `P` of its centre.

1.1 Forces and response to them

The dynamic on sets acts by moving each point in a set, subject to two kinetic rules:

- If movement puts two points into the same xel, they merge.
- If points p and q cease to be neighbours, a new point appears in the xel between them, on the line $\overline{pq}$.

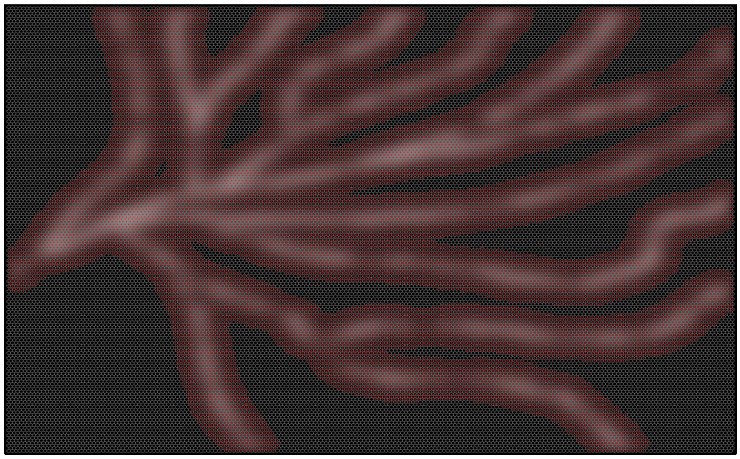
By the first rule, the changed set remains a xelset, so the rule holds in all implementations. By the second, its connectedness does not change: to model a shape that can break, the rule must be appropriately modified.

The points are moved by two kinds of force: intrinsic to the xelset, and extrinsic. In the current application the intrinsic forces resemble surface tension, pushing each point toward sharing a straight line with its neighbours. We omit the mathematics here, noting only that the algorithm is contained in the function `bufferNewPoints()`.

The extrinsic forces come in the current version from a scalar (grey-levels) image $\mathcal{G}$, pushing the point 'uphill' toward brighter points. The image $\mathcal{G}$ is assumed to be derived from an image $\mathcal{I}$ such as those in Fig. 2, by image processing that produces high values at points that are likely to be high on a visible curve. For the lightning photograph we could simply use the red or green channel, but the fundus image requires much more processing to distinguish the arteries from the background. A fully developed application will include a pipeline from $\mathcal{I}$ to $\mathcal{G}$ to xelular dynamics to the export and analysis of the network, extracting features of medical interest, but the package here described simply imports $\mathcal{G}$ as a bitmap.

To find the edge of the optic disc (the bright area where the arteries enter the eye), one can use an edge-detecting numerical filter. The curve along which it is bright could be found by a xelset, or (being topologically simple) by a classical 'snake' loop. For the branching blood vessels — our first application — we will drive a xelset to bright ridges in an image $\mathcal{G}$ found by a use of Gabor filters, including some new ideas to be described elsewhere. It is worth mentioning here that the $\mathcal{G}$ with grey-scale pixel values traditional in such image analysis will be replaced by a *vector* at each pixel, adding a direction to the intensity of 'ridge-like-ness' at each pixel. This will require some adjustment to the data structure for $\mathcal{G}$, and allow the curves created by a xelset to be most attracted by intensity in the right direction, by modifying the gradient forces encoded in `bufferNewPoints()`.

(a)



(b)

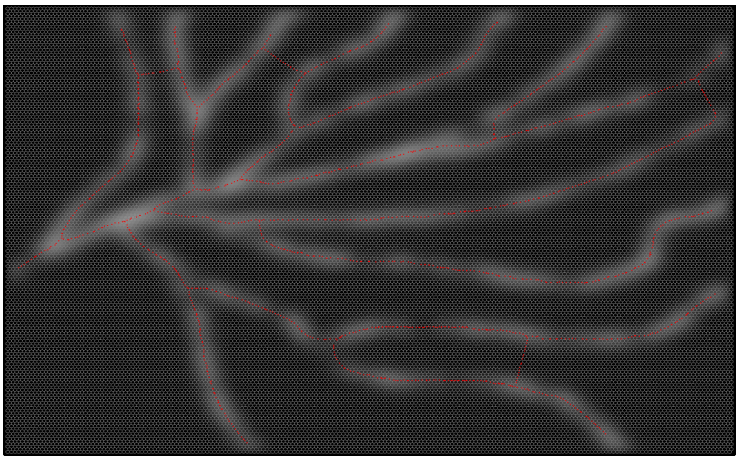


Figure 5: a) A grey-level image $\mathcal{G}$, with a starting 'blob' xelset $\mathcal{X}$ of points where $\mathcal{G} > \mathcal{E}$; b) The set to which $\mathcal{X}$ shrinks.

Gaussian blurring of the image $\mathcal{G}$ reduces the effect of noise, and allows gradient forces to point smoothly 'up-hill' toward the bright ridges. It also extends the influence of the bright places across the dark regions between them. The forces point toward the bright ridges over a greater distance, guiding the xelset points more appropriately. (This is less critical for xelsets than for snakes. A snake in a uniformly dark region has no reason to move sideways, one way or the other, but intrinsic forces on their own move the boundary of a xelular blob into the set.) In Fig. 5a we initialise the xelset $\mathcal{X}$ with the centre of every xel where the grey level from $\mathcal{G}$ is above a chosen threshold $\mathcal{E}$, whichh leaves useful holes which grow in $\mathcal{X}$, as it shrinks to a clearly branched structure (Fig. 5b). Using an un-blurred version of $\mathcal{G}$ at this point could create too many small holes by the thresholding process. However, once the topology is correct, it is useful to replace the blurred $\mathcal{G}$ by a less-blurred version, attracting $\mathcal{X}$ more sharply to ridge points that correspond to the true target. This cycle of blurring levels is not implemented in the current code.

As Fig. 5b illustrates, the shrinkage of $\mathcal{G}$ can produce 'cross-links' which do not follow a ridge of high grey levels. The threshold initialisation of of $\mathcal{X}$ may create two holes where only one is needed, and the thick bridge between them becomes an unwanted curve, usually straight since image forces are small. Such a link can easily be removed,

after the step of extracting a network description of the set to which $\mathcal{X}$ shrinks. This step requires creating a list of ends and branch points (graph vertices), and a list of curves (graph edges) which identifies for each curve $\mathcal{L}$ the vertices where it begins and ends, and the points (x, y) it includes between them. If the average brightness of $\mathcal{G}$ at 'between' points of $\mathcal{L}$ is low, remove it. (It may be best to perform this step using a less-blurred $\mathcal{G}$: not yet implemented.)

The list structure of nodes and branches, after the removal of unwanted links, is the main output of the current code, aside from the interaction process described below. Data with such a structure will be used for further analysis of medically significant properties such as tortuosity, oxygenation, and other quantities not covered in the current code.

2 Xelset demonstration package

The C++ code package `xel2d` can run on the Mac OSX and Linux operating systems, with the OpenGL library installed. A user can compile the included makefile `/xel2d/make`, which is common for both mac and linux. The compilation produces the executable file `/xel2d/xel2d.out`. Rather than using command-line arguments, `xel2d` is controlled by a few parameters, set in the code file `/xel2d/src/param.cpp`, discussed below as the effect of each becomes relevant.

The program's running can be divided into three stages:

1. Initialization of xelset
2. Evolution of xelset
3. Extraction of network

explained in the following sections.

2.1 Initialization of xelsets

The program can run in two different modes.

1. Interactive mode
2. Image mode

Initialization of the xelset sets up the data of which xels contain points of the set, and the coordinates of these points. It depends here on which mode the program is running. A user chooses a mode by setting up the parameter `programMode` in `/xel2d/src/param.cpp`. Setting it to '0' directs the program to run in Interactive mode, while '1' directs it to Image mode.

2.1.1 Interactive Mode

In interactive mode, the program initially shows the user an empty 2-dimensional grid (Fig. 6a) of hexagons, indicating xels. Its dimensions are set by the integer parameters `Nx` and `Ny` in `/xel2d/src/param2d.cpp`, with every hexagon's side length equal to the integer parameter `xelSize` in pixel-width units. Only intrinsic forces are applied, so this is a setting in which to explore their effects. If no points are made static, the xelset simply shrinks to an isolated point (or points). When they are present, its shrinkage creates a network between them.

The user inputs a xelset point by point. A click at any point in the xel-grid enters that point in the corresponding xel, if no point has been created in that xel before. A left click creates a mobile point, displayed in red, while a right click create a static point, displayed in yellow: see Fig. 6b. A middle click in an occupied hexagon (if the scroll wheel is clickable), removes the point from the corresponding xel. Pressing 'f' adds a point at the centre of every empty xel. creating a maximal blob: in Fig. 6c this has been done with the same static points as in Fig. 6b. Iterative shrinking creates the set shown in Fig. 6d.

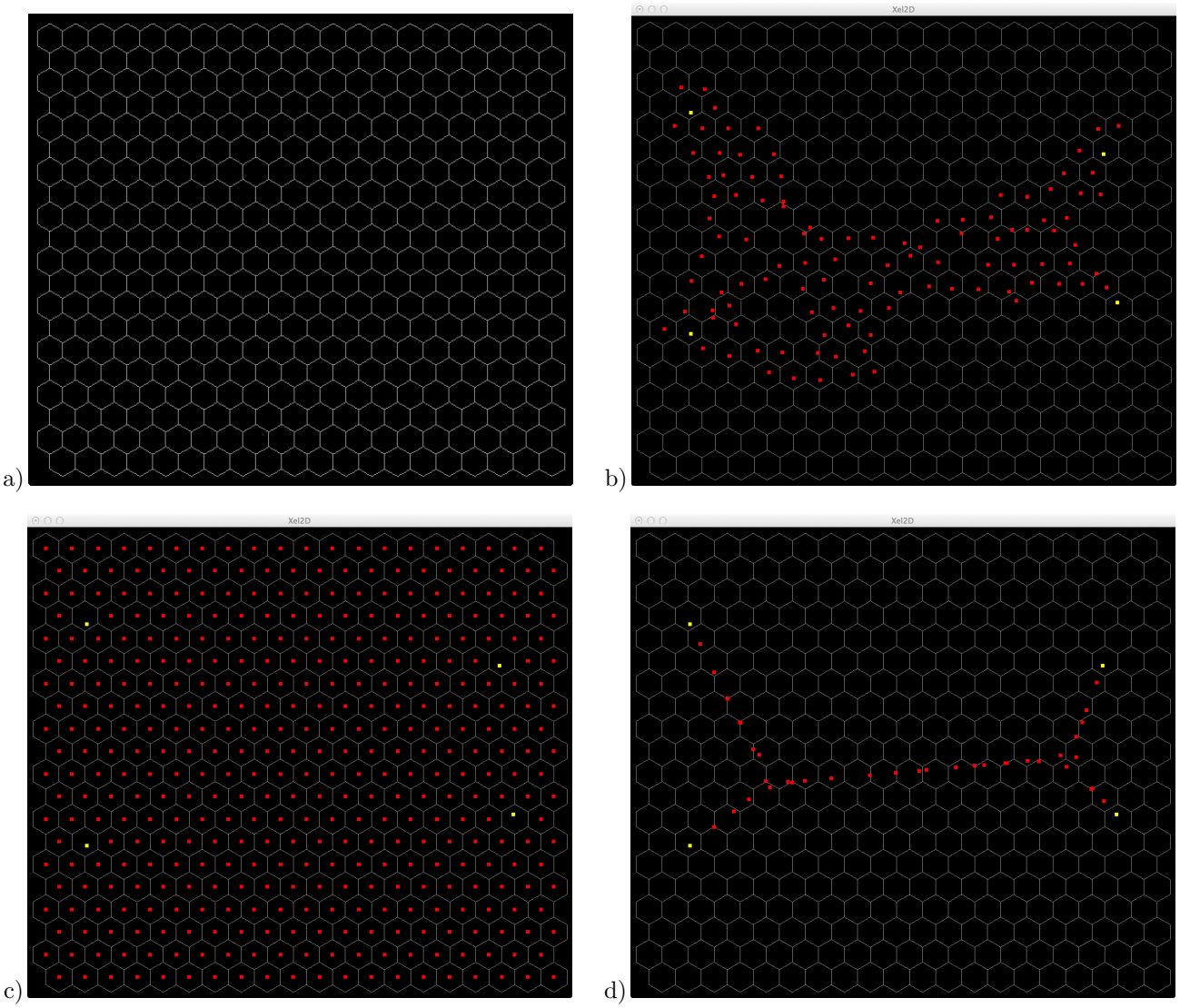


Figure 6: **(a)** empty xel grid; **(b)** xel set with four static points and a hundred mobile points; **(c)** all xels filled, four of them static; **(d)** what (b) or (c) shrinks to.

2.1.2 Image Mode

In Image mode, no points are static; since a xel is either empty or occupied by a mobile point, for pure image-driven code the attribute `xelType` could be replaced by a boolean 'occupied' and many `if`s simplified.

In this mode, parameter `xelSize` affects only the OpenGL display of the computation, not the xelular resolution itself: setting it to 5, then dragging the initial display to double the width, gives exactly the same image as setting it to 10 in the first place. (It would be possible to use hexagons that are larger relative to the image, for faster computation and a coarser final network fit, but this is not implemented here.)

The program starts by reading a bitmap image such as Fig. 7a, whose location and name are set by the parameter `inputFile` in the file `/xel2d/src/param2d.cpp`. The code assumes it to be a 24-bit RGB file, and converts it to grey scale, by the usual photographic formula. (If it is a grey-looking R=G=B image, choosing any one channel would have the same effect.) Then a series of `blurCount` Gaussian blurring steps reduces the local complexity (Fig. 7b) and allows a gradient operator to point toward regions of greater intensity, and push points in that direction.

Next, the program creates a copy of these grey values, resampled on a hexagonal grid whose dimensions are set according to the image size and `xelSize` (the hexagon side length), with as many hexagons in a horizontal row as

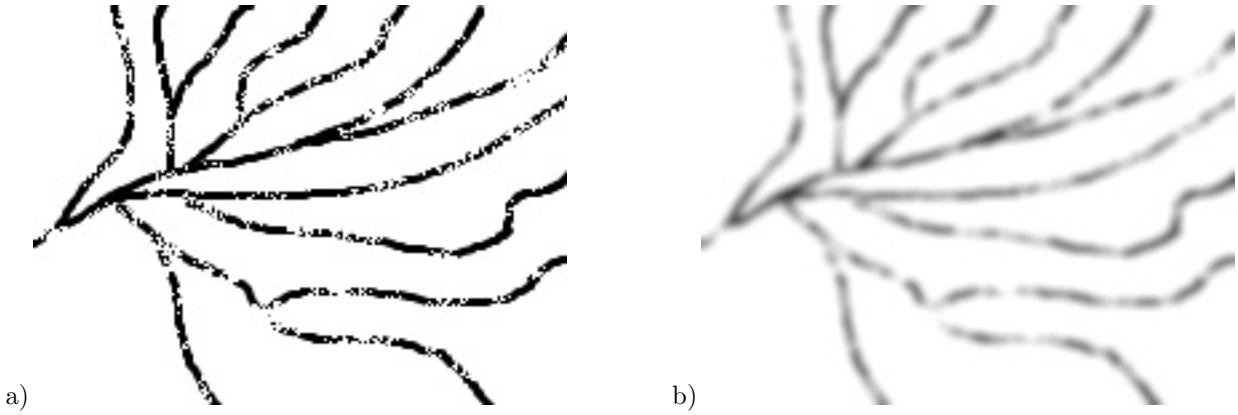


Figure 7: **(a)** a sample input image; **(b)** the same image, smoothed by blurring.

the pixel width of the image. Because in a hexagonal array the vertical spacing between rows is only $\sqrt{3/4}$ the spacing of points along a row, there are approximately $\sqrt{4/3} \approx 1.155$ more rows in the xel array than there are rows in the image, and thus more points by the same factor. An empty grid of xels is created with these dimensions, so that each xel correspond to a resampled grey value. If this value exceeds the floating-point threshold parameter $\mathcal{E} = \texttt{imageThreshold}$, a central point is added to the corresponding xel. The final hexagonal grid is drawn, with the resampled grey levels, and xelset points in red.

2.2 Evolution of xelsets

Once the xelset is initialised, according to either program mode, the user can use the keyboard to control its evolution: pressing 'i' executes one iteration step, while 'p' starts or stops an indefinite sequence of steps.
At any time, the user can also press 'f' to fill all the empty xels of the xel grid.

2.3 Extraction of the network

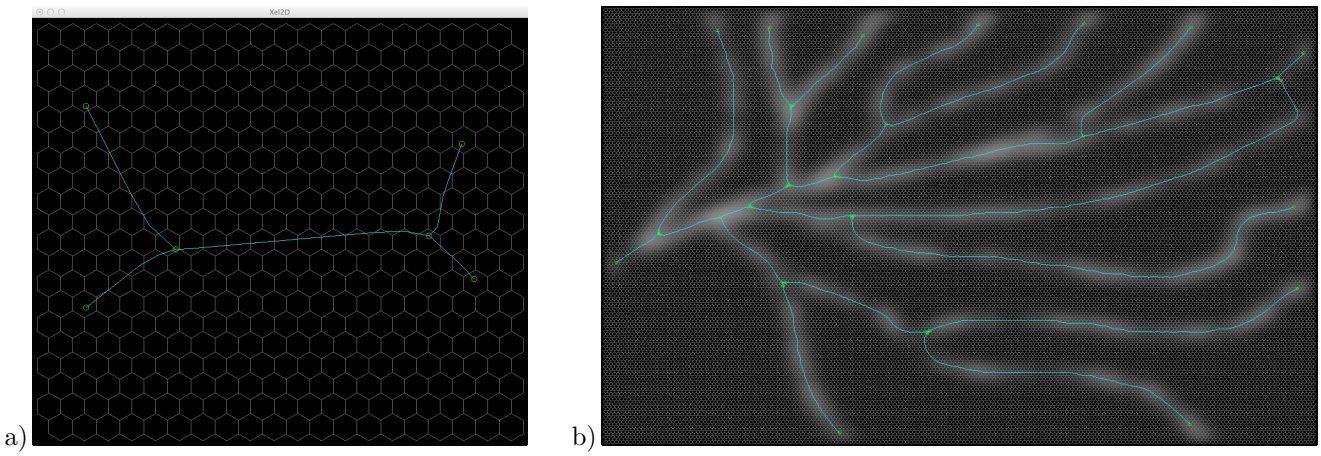


Figure 8: a) The network extracted from Fig. 6d; b) The network extracted from Fig. 5 b, whose input was Fig. 7.

The final stage is to extract network information from the shrunken xelset, by pressing 'n' on the keyboard. The view then changes to show nodes and branches, as shown in Fig. 8a for the Interactive Mode example in Fig. 6d, and in Fig. 8b for the Image Mode example in Fig. 5 More details are in §4.9 below.

The display can be returned to the default view, where the 'i' and 'p' keys are active, by pressing 'n' again. Pressing 'n' is useful only when the xelset has shrunk to a network of curves, since there is a simplification step

that assumes this has happened. Invoking the network extraction while it has blobs of points gives absurd results.

A user is not bound to execute these stages in a strict sequence. One can run a few more iterations after the network is extracted, and then perhaps extract the network again. In Interactive mode, the user can add or remove points at any time when the process is not active and the display shows the points, not the network.

3 Rows and neighbours

Hexagons fit less comfortably than squares do into a rectangle (such as we usually need to cover to interact with an image): if we have horizontal rows, we cannot have vertical columns. We deal with this using shaky columns — vertical on average — but this in turn makes creates a little complication in referring to points in the immediate neighbours of a xel χ, as need to in computing the intrinsic forces on a point in χ.

For iteration over all xels, rather than a two-index `Nx`×`Ny` array we declare a 1D array of length `Nx*Ny`. This requires a function `to_iI(int i)` which returns a 2D location `I`, and its inverse `to_Ii(TXY<int> I)`; the constant value of the 2D index location `I`, and the 2D geometric location (x, y), are computed once for each xel by the constructor and stored, at a small memory cost but no repeated evaluation. We use `I` in combination with `nbrI[]` to find the neighbours of the `i`$^{\text{th}}$ xel.

3.1 Positions of xel centres

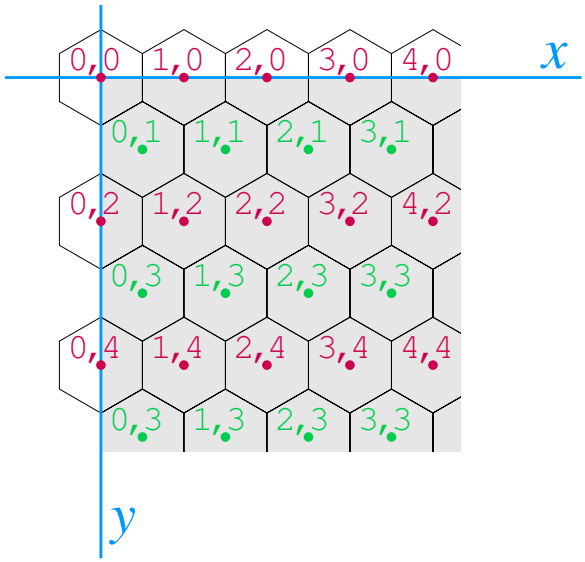


Figure 9: The 2-index labels of xels, as located in the plane.

We label xels `I`=(`i`, `j`) by row `j` (counting downward, from 0), and position `i` in the row (left to right, from 0). (The 'downward' matches the graphics habit of downward-y, which originated in the scanning order of CRT hardware.) In geometric (x, y) position, the first xel is at the origin. From Fig. 4 it is clear that within a row the xel centres are `pitchX`=2`*minorRadius` apart, while the rows are `pitchY`=$\frac{3}{2}$`*majorRadius` apart. For `j`-even-numbered rows, the centre of a xel with `I`=(`i`, `j`) is thus at (`pitchX*`x, `pitchY*`y). However, an `j`-odd-numbered row is spatially offset by `minorRadius`, so that the centre of a xel in it is at (`minorRadius`+`pitchX*`x, `pitchY*`y). Combining these, and using the C "`%`" notation for "modulo", we have in general

$$(x, y) = \Big(\texttt{(j\%i)*minorRadius+pitchX*}x,\ \texttt{pitchY*}y\Big)\,,$$

as implemented in both versions of `getXelMidPoint()` in `xel2d.cpp`.

More subtly, the next function `getXelIndex()` in `xel2d.cpp` takes as argument a general plane point (x, y) (which need not be a xel centre), and returns the `I`-style double index of the xel that point is in, by a quick use of dot products. *(Some time: dig out original text and figures for this algebra; patch in here.)* This is important in deciding which xel a moved point is now in.

3.2 Neighbouring xels

Unlike a square array, where the four nearest neighbours of (`i`, `j`) are always (`i`$\pm$`1`, `j`) and (`i`, `j`$\pm$`1`), listing the neighbours of xel (`i`, `j`) involves the parity of `i`. As Fig. 9 makes clear, the even-`j` neighbours of a point (`i`, `j`) for which `j` is odd have positions (`i`,`j`$\pm$`1`) and (`i+1`,`j`$\pm$`1`) . The odd-`j` neighbours of a point (`i`, `j`) with `j` even have positions (`i-1`,`j`$\pm$`1`) and (`i`,`j`$\pm$`1`). In both cases the neighbours in the same row are (`i`$\pm$`1`,`j`).

Rather than a C-code formula to encapsulate these cases, we handle this by using the global variable array of pairs `nbrI[12]`, declared in `xel2d.cpp` and initialised by the constructor of the first `TXel2D`; the first six members give the (`i`, `j`)-relative neighbour positions for an even `i`, the other six for an odd `i`. Then the pair `nbrI[k+(j%2)*6]` gives the `k`$^{\text{th}}$ pair from the first six when `j` is even, the `k`$^{\text{th}}$ pair from second first six when it is odd. This array is used in several places to iterate over neighbours.

Boundary xels, of course, have fewer neighbours, so those cases must be checked for. Overall the resulting code is a little opaque, but behaves stably, and is unlikely to need change as long as we use rectangular images.

4 Code Structure

This section discusses the details of code organization and program flow. It is mainly for developers who want to understand and work further with the code. Following it step by step while referring to the `xel2D` code should enable a developer to have a strong grip on the code. It is recommended to follow it in a sequential way. Jumping sub-sections may create confusion.

4.1 Directory Structure

The `xel2D` code is packaged in the root folder `xelSets` which can be cloned from the link : `'https://github.com/badalindia/xelSets'`. The root folder has the following hierarchy.

- `opengl/`
  - `osx`
  - `linux`
- `xel2d/`
  - `src/`
  - `obj/`
  - `makefile`
  - `xel2d.out`

The root folder contains two folders: `opengl` and `xel2d`. The folder `opengl` contains necessary C++ OpenGL header files, for both linux and mac operating systems, in the separate folders `linux` and `osx` respectively. The folder `xel2d` has two folders (`src` and `obj`) and two files (`makefile` and `xel2d.out`). The folder `src` contains all the C++ source code files, and `obj` contains the intermediate objective files created and managed by the compiler. The file `makefile` is used to compile the program in either Linux or OSX. Compiling it successfully creates the executable file `xel2d.out`, which runs the `xel2d` program.

4.2 Source Files

As mentioned in §4.1, all the source files are present in the directory `/xel2d/src`. They are listed below and discussed in the following sections as needed.

- `xel2dmain.cpp`
- `param2d.h & param2d.cpp`
- `point2d.h & point2d.cpp`
- `xel2d.h & xel2d.cpp`
- `input2d.h & input.cpp`
- `uihandler2d.h & uihandler2d.cpp`
- `xeltree2d.h & xeltree2d.cpp`
- `bitmap_image.h`

4.3 Entry Point of the Program

Like any generic C++ program, `xel2d` starts with the `main()` function defined as `'int main()'` in the file `/xel2d/src/xel2dmain.cpp`. The code is as follows:

```
1  //main
2  int main()
3  {
4    //create instance of TXel2D
5    TXel2D *xel;
6
7    //initialize xel grid
8    if(programMode == 0)
9    {
10     xel = new TXel2D[Np];
11   }
12   else
13   if(programMode == 1)
14   {
15     loadImage();
16     xel = new TXel2D[Np];
17     resampleImage(xel);
18   }
19
20   //intialize OpenGL interactive session
21   initUI(xel);
22 }
```

Listing 1: The function `main()`.

Fig. 6 illustrates the xel grid, with each hexagon representing a xel, and the point in each non-empty xel — *not* usually at the centre — shown in red or yellow. The data associated to one xel is encapsulated in a class called `TXel2D` defined in the header file `/xel2d/src/xel2d.h`, explained in detail in §4.4.3. A xel grid is a 2D array of the datatype `TXel2D`. The first thing that the function `main()` does, in line 5 of code listing 1, is to create a reference of a dynamic array, `xel`, of type `TXel2D`. Next, it initializes the array according to the program mode (as discussed in §2.1). In interactive mode, it initializes `xel` with an array of `Np` elements of type `TXel2D` (line 10 of listing 1). This linear array stores row-wise the information of a 2-dimensional xel grid. In image mode, line 15 calls the function `loadImage()`, whose working is explained in §4.5. For now, it suffices to know that this function loads the image found at the file address in the parameter `inputFile`, converts it into grey scale data, maintains a copy somewhere,

and sets the parameters `Nx`, `Ny`, and `Np` (dimensions of xel grid) according to the size of the image loaded. Then `main()` initializes `xel` with an array of size `Np`, in the same way as for interactive mode. In line 17 of listing 1 the function `resampleImage()` inserts points in the entries of the array `xel` (sent to it as an argument) according to the image data. §4.6 gives a detailed account of this function.

At this point, the xels have been initialized. Lastly (line 21 of listing 1), `main()` calls the function `initUI()`, initializing an OpenGL interactive session which handles all the further flow of program, which includes xelular evolution and network detection. In interactive mode, it also allows the user to add or remove points in the xelset, as discussed in §2.1.1. The details of this OpenGL interactive session are discussed in §4.7.

4.4 Essential Classes and Code Elements

To explain the further functionality of the `xel2d` code, we introduce a few essential structures and elements that are frequently used throughout.

4.4.1 Program Parameters: param2d.h

All the user input parameters upon which the program depends are defined in the header file `param2d.h`.

```
1  //program parameters
2  extern int Nx, Ny;
3  extern long Np;
4  extern int programMode;
5  extern string inputFile;
6  extern int blurCount;
7  extern float imageThreshold;
8  extern const int xelSize;
9  extern const float minorRadius, majorRadius;
10 extern const float pitchX, pitchY;
11
12 //generic macro for FOR loops
13 #define FOR(i,N)  for(int i=0;i<N;i++)
```

Listing 2: Parameters in `param.h`

All the parameters are declared as `extern` variables, and can thus be accessed in any file which includes `param2d.h`. The integers `Nx` and `Ny` are respectively the number of xels in each row of the xel grid, and the number of rows, while $\texttt{Np} = \texttt{Nx} \times \texttt{Ny}$ is the total number of xels. The parameter `programMode` determines whether the program runs in interactive mode or image mode. In image mode the address of the image file which is required to be loaded is set to the parameter `inputFile`. The parameter `blurCount` determines how many times the image data has to be blurred. The parameter `imageThreshold` is used for thresholding intensities (as illustrated in Fig. 5). The values of these parameters are set in `param2d.cpp`, as are some derived quantities (see Fig. 4).

Apart from parameters, `param2d.h` defines a macro short-hand for '`for`' loops, shown in line 13 of listing 2.

4.4.2 TXY

Class `TXY` is a template class defined in file `point2d.h` to store the information of 2 dimensional points. The class definition is as follows:

```
1  template<typename T>  class TXY
2  {
3  public:
4    T x, y;
5
6    //constructors
7    TXY();
```

```
8    TXY(T x, T y);
9
10   //functions for setting x & y values
11   void set(T x, T y);
12   void set(T v);
13
14   //returns the length of position vector : sqrt(x^2 + y^2)
15   float length();
16
17   //prints the 2D point coordinates on terminal
18   void print();
19
20   //operator overloading
21   TXY<T> operator+(TXY<T> ob1);
22   TXY<T> operator-(TXY<T> ob1);
23   TXY<T> operator*(T val);
24   TXY<T> operator/(T val);
25   void operator=(TXY<T> ob1);
26   bool operator==(TXY<T> ob1);
27 };
```

Listing 3: The class `TXY`

The class is implemented in the file `point2d.h`, in a self explanatory way. It has two data members `x` and `y`, of type `T`, which must be a numerical data type. There are two constructors, and two member functions to set the values of data members. The function `length()` returns the length $\sqrt{x^2 + y^2}$ of the vector (x, y). The function `print()` just displays on the console the values of the data members.

The main purpose of this data type is to overload several arithmetic operators to apply to 2D vectors. Points of the same type interact in a simplified way, as in Listing 4 below:

```
1 TXY<float> p1(1,2), p2, p3, p4;
2 p3 = (p1 + p2)*0.5;
3 p4 = (p1 == p3)?p1:(p1 + p3)*0.5;
```

Listing 4: Usage of class `TXY`, finding the mid-point of `p1` and `p2`, and of `p1` and `p3`.

4.4.3 TXel2D

As introduced earlier, the class `TXel2D`, defined in the file `/xel2d/src/xel2d.h`, encapsulates all the data associated to a xel, along with some functions that operate on it. Its definition is as follows:

```
1  class TXel2D
2  {
3  public:
4    const TXY<int> I;           //Index of xel in the 2d xel grid
5    const TXY<float> P;         //mid point of xel
6
7    TXY<float> p;               //floating point position
8    int xelType;                //0 = empty, 1 = static, 2 = mobile
9    int networkType;            //type of point in xel network
10
11   float Y;                    //sampled grey value from the image
12
13   TXY<float> departingPoint; //the point after movement
14   TXY<int> departingXel;     //the xel index of point after movement
15   TXY<float> enteringPoint;  //point to be entering after movement
16   int enteringPointCount;    //count of the entering points
```

```
17
18    TXel2D();                    //constructor
19    ~TXel2D();                   //destructor
20 };
```

Listing 5: The class `TXel2D`

A `TXel2D` is always a member of a xel grid, in which it has a two-dimensional index. That index is stored in the member '`I`' of class `TXel2D`. The centroid of each xel is stored in its member '`P`'. Both these members are constant because they do not change once the xels are initialized. The member '`p`' stores the location of the point in the xel, if there is one present. The member '`xelType`' stores the state of the xel: '0' for empty, '1' for containing a static point, and '2' for a mobile point. The member '`networkType`' stores information related to network extraction, discussed in §4.9. The member '`Y`' stores a grey level value associated to the xel, calculated by resampling the image data onto a hexagonal lattice (as discussed in §4.6). The other members '`departingPoint`', '`departingXel`', '`enteringPoint`', and '`enteringPointCount`', used in the process of xel evolution, are described in detail in §4.8.

4.5 Loading the input Image: the function `loadImage()`

This function is prototyped in the header file `/xe2d/src/input2d.h` and defined in the file `/xe2d/src/input2d.cpp`. It is called from `main()` at the beginning of the program to load the image data (§4.3). Its body is as follows:

```
1  void loadImage()
2  {
3    //image object
4    bitmap_image image(inputFile.c_str());
5
6    //dimensional init
7    imageWidth = image.width();
8    imageHeight = image.height();
9    long pixelCount = imageWidth*imageHeight;
10
11   //extracting image data and converting it to grey scale
12   imgR = new float [pixelCount];
13   imgG = new float [pixelCount];
14   imgB = new float [pixelCount];
15   imgY = new float [pixelCount];
16   image.export_rgb(imgR, imgG, imgB);
17   FOR(i, pixelCount)
18   {
19    //conversion to grey image
20    imgY[i] = 0.3*imgR[i] + 0.59*imgR[i] + 0.11*imgB[i];
21    imgY[i] = 1 - imgY[i];
22   }
23
24   //bluring
25   blurY(imgY, imageWidth, imageHeight, blurCount);
26
27   //calculating xel grid size
28   Nx = imageWidth;
29   Ny = imageHeight*pitchX/pitchY;
30   Np = Nx*Ny;
31 }
```

Listing 6: The function `loadImage`

The function reads and writes bitmap images by use of the library `bitmap_image.h`, present in the folder `/xel2d/src/`. The details of this library, available at http://www.partow.net, are beyond our scope here.

The function `loadImage` first creates an object `image` of class `bitmap_image` (defined in `bitmap_image.h`), used as an interface for reading the image. Line 4 of listing 6 sends the image file address, stored in the parameter `inputFile` of C++ type `string`, to the constructor of `bitmap_image.h`, as a `char` array found by the call `inputFile.c_str()`. To store the image data, the variables `imageWidth`, `imageHeight`, `imgR`, `imgG`, `imgB`, and `imgY` are declared as `extern` variables in the header file `xel2d.h`, and so can be accessed (read or written) from any file which includes it. Lines 7 and 8 of the function `loadImage` set the values of `imageWidth` and `imageHeight` respectively, and line 9 stores the total pixel count as a variable named `pointCount`. Lines 12 to 15 initialize `imgR`, `imgG`, `imgB`, and `imgY` with arrays of size `pointCount`. In line 16, the function `image.export_rgb()` writes image data from the red, green and blue bands respectively to these arrays. Next, a loop converts all RGB values to grey levels stored on the array `imgY`. Line 21 inverts the grey level, exchanging the extremes '0' and '1' and '1' for clarity of display. Line 25 blurs the image by a call to the function `blurY()` (declared in `input2d.h`),which takes the arguments: the array to be blurred (`imgY`), image width (`imageWidth`), image height (`imageHeight`), and the number `blurCount` of times to apply a Gaussian convolution, a parameter set in `param2d.cpp`. Lastly, lines 28 to 30, calculate the dimensions of the xel grid (`Nx`, `Ny`, and `Np`) from those of the image.

4.6 Re-sampling Image Data: `resampleImage()`

The function `resampleImage()` is prototyped in the header file `/xe2d/src/input2d.h` and defined in the file `/xe2d/src/input2d.cpp`. Called from `main()` once the image is loaded and the xel grid initialized, it resamples the image data onto the hexagonal lattice of xel centres. Any xel with a grey level intensity more than `imageThreshold` acquires a point at its centre. The body of the function is as follows:

```
1  void resampleImage(TXel2D *xel)
2  {
3    //space between pixels (pixel-pitch) is assumed to be equal to pitchX of xel-space
4    FOR(i, Np)
5    {
6      TXY<float> p = xel[i].P;
7      p = p/pitchX;
8      TXY<int> p00(floor(p.x), floor(p.y));
9      float y00 = (p00.x>=0 && p00.x<imageWidth && p00.y>=0 && p00.y<=imageHeight)?imgY[
           p00.y*imageWidth + p00.x]:0;
10     float y01 = (p00.x+1>=0 && p00.x+1<imageWidth && p00.y>=0 && p00.y<=imageHeight)?
           imgY[p00.y*imageWidth + p00.x+1]:0;
11     float y10 = (p00.x>=0 && p00.x<imageWidth && p00.y+1>=0 && p00.y+1<=imageHeight)?
           imgY[(p00.y+1)*imageWidth + p00.x]:0;
12     float y11 = (p00.x+1>=0 && p00.x+1<imageWidth && p00.y+1>=0 && p00.y+1<=imageHeight)
           ?imgY[(p00.y+1)*imageWidth + p00.x+1]:0;
13     TXY<float> dp(p.x - p00.x, p.y - p00.y);
14     xel[i].Y = (1-dp.y)*(y00*(1-dp.x) + y01*(dp.x)) + dp.y*(y10*(1-dp.x) + y11*(dp.x));
15     if(xel[i].Y > imageThreshold)
16       addMidPointToXelSet(xel, xel[i].I, 2);
17   }
18
19 }
```

Listing 7: The function `resampleImage`.

It takes as argument a reference to the `TXel2D` array `xel`, containing the xel grid data. Line 4 of listing 7 initiates a loop that goes `Np` times, once for each xel in the xel grid. For each xel, line 7 scales the (x, y) coordinates of its mid-point 'P' into pixel units, and lines 9–12 determine the four surrounding points of the image's square lattice. From the grey levels at these points, 14 performs a standard bilinear interpolation to get a value at `P`, stored with the xel. The remaining two lines test this value, and add a point if appropriate.

4.7 Initializing OpenGL: initUI()

The function `initUI` is called from `main()` once the xel structure has been initialized. It is declared in `uihandler2d.h` and defined in file `uihandler2d.cpp`. The body of the function is as follows:

```
1  void initUI(TXel2D *xel)
2  {
3    //keeping a global copy of structure
4    ::xel = xel;
5
6    //dummy arguments
7    int argc = 0;
8    char** argv;
9
10   //dimensional parameters
11   offsetX = minorRadius + 10;
12   offsetY = majorRadius + 10;
13   windowX = (Nx - 1)*pitchX + offsetX*2.0 + minorRadius;
14   windowY = (Ny - 1)*pitchY + offsetY*2.0;
15
16   //initiating hexagonal sides
17   FOR(i, 6)
18   {
19     xelHex[i].set(xelSize*cos((30 + 60*i) RADIAN), xelSize*sin((30 + 60*i) RADIAN));
20   }
21
22   //glut init
23   glutInit(&argc, argv);
24   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
25   glutInitWindowSize(windowX,windowY);
26   glutInitWindowPosition(0,0);
27   glutCreateWindow("Xel2D");
28
29   //projection
30   glMatrixMode(GL_PROJECTION);
31   glLoadIdentity();
32   glOrtho(0, windowX, windowY, 0, 0, 1);
33   glMatrixMode(GL_MODELVIEW);
34
35   //Setting Default Parameters
36   glEnable(GL_COLOR_MATERIAL);
37   glClearColor(0.0, 0.0, 0.0, 0.0);
38
39
40   //assigning handler functions
41   glutDisplayFunc(drawHandler);
42   glutKeyboardFunc(keyHandler);
43   if(programMode == 0)
44     glutMouseFunc(mouseHandler);
45   glutIdleFunc(process);
46
47   //initiate gl main loop
48   glutMainLoop();
49 }
```

Listing 8: The function `initUI`

It takes the `TXel2D` array, `xel`, as a parameter containing the xel grid data. It stores the reference to this array as a local copy, recognizable by the other functions defined in `uihandler2d.cpp` (line 4 of listing 4.7). Lines 7 and

8 declare two dummy variables, `argc` and `argv`, and initialize them with null values. Line 23 uses them to call the function `glutInit`, defined in OpenGL library: this expects command line arguments as parameters, but we don't necessarily need to do that. Lines 11 to 14 initialize the parameters `offsetX`, `offsetY`, `windowX` and `windowY`, which ultimately depend on parameters defined in `param2d.h`. The parameters `offsetX` and `offsetY` are used later for drawing purposes, while `windowX` and `windowY` define the size of the OpenGL display window. Lines 17 to 20 set an array of 6 points to the vertex positions in a regular hexagon of the current size, with respect to its center. Line 23 to 26 call OpenGL library functions to initialize Opengl and set a display window. (Understanding their significance requires a grasp of OpenGL.) Lines 29 to 37 set some OpenGL parameters elated to the display, and lines 41 to 45 bind OpenGL event handler functions to OpenGL events. The function `drawHandler()` is set as the display function, called whenever the OpenGL needs to draw. All the display routine of drawing hexagons, points, *etc.*, are implemented within this function. The function `keyHandler()` is set as the keyboard event function: called whenever a key is pressed, it receives the key's identity and acts accordingly. The function `mouseHandler()` is set (if the program is running in the interactive mode) as the event function which is called whenever the mouse is clicked. It implements the interface of allowing user clicks to add or remove points from the xel grid. The function `process()`, set as the default function that is called repeatedly when nothing else is happening, accounts for the background processing that happens in OpenGL while the user is allowed to interact with the OpenGL display. Lastly, the function `glutMainLoop()` calls for the OpenGL display to appear and begin the interaction session. At this point, the control flow jumps from sequentially following function calls to an OpenGL event driven model.

4.8 Evolution of Xel Sets

The OpenGL interactive session allows user to do many things, such as to execute one or many evolution steps. A single iteration of an evolution step is implemented in the function `cpuRun()` declared in `xel2d.h` and defined in `xel2d.cpp` (code listing 9). If the user presses the key '`i`', during the OpenGL interactive session this function is called once. And, if the user presses the key '`p`', it is called repeatedly until '`p`' is pressed again.

```
1 void cpuRun(TXel2D *xel)
2 {
3   bufferNewPoints(xel); //calculate movement, apply and buffer new positions
4   interpolatePoints(xel); //interpolate points between new buffered positions
5   finishIteration(xel); //write new points, delete old
6 }
```

Listing 9: The function `cpuRun()`.

The function `cpuRun()` takes the `TXel2D` array, `xel`, as an argument. Each iteration is broken into a series of three steps, implemented in three functions explained in the following three sub-sections.

4.8.1 The function `bufferNewPoints()`

Both intrinsic and extrinsic forces on each point are computed in this function, and a moved position computed accordingly. (If this is in a different xel, the results there must be handled.)

After the constructor for `TXel2D`, including the parity-dependent data for neighbour handling, this function defines the functions connecting the single-index and two-index ways to refer to a xel, some self-explanatory geometric and re-set functions, and the `cpuRun()` mentioned above. Then comes `bufferNewPoints()`, containing the core xelular logic for both intrinsic and extrinsic forces, whose listing follows.

```
1  void bufferNewPoints(TXel2D *xel)
2  {
3    FOR(i, Np)
4    {
5      //calculate the new moved point if the xel is 'movable'
6      if(xel[i].xelType==2)
7      {
8        //xel force
9        TXY<float> xelF(0.0, 0.0);
10       //image force
```

```
11        TXY<float> imgF(0.0, 0.0);
12
13        //nbr count and mean point calculation
14        TXY<int> I(0,0);
15        int zMod2 = xel[i].I.y % 2;
16        int dj = zMod2*6;
17
18        TXY<float> mean(0, 0);
19        int nbrCount = 0;
20
21        //calculating mean of neighbors
22        FOR(j,6)
23        {
24          I = nbrI[dj+j] + xel[i].I;
25          if(INRANGE(I))
26          {
27            int in = to_Ii(I);
28            if(xel[in].xelType==1 || xel[in].xelType==2)
29            {
30              mean = mean + xel[in].p;
31              nbrCount++;
32            }
33          }
34        }
35        mean = mean/nbrCount;
36
37        //calculating image force
38        if(nbrCount < 6)
39        {
40          TXY<float> _p = xel[i].p/pitchX;
41          int X1 = floor(_p.x);
42          int Y1 = floor(_p.y);
43          if(X1>=0 && X1<imageWidth-1 && Y1>=0 && Y1<=imageHeight-1)
44          {
45            float dy = _p.y - Y1;
46            float dx = _p.x - X1;
47            imgF.x = (1-dy)*(imgY[Y1*imageWidth+X1+1] - imgY[Y1*imageWidth+X1]) + dy*(imgY
                  [(Y1+1)*imageWidth+X1+1] - imgY[(Y1+1)*imageWidth+X1]);
48            imgF.y = (1-dx)*(imgY[(Y1+1)*imageWidth+X1] - imgY[Y1*imageWidth+X1]) + dx*(
                  imgY[(Y1+1)*imageWidth+X1+1] - imgY[Y1*imageWidth+X1+1]);
49            imgF = imgF * imageForceScale;
50          }
51        }
52
53        //calculating xelular force
54        if(nbrCount>1 && nbrCount<6)
55        {
56
57          //generating inertial tensor
58          float xx=0, xy=0, yy=0;
59          TXY<float> dp;
60
61          FOR(j,6)
62          {
63            I = nbrI[dj+j] + xel[i].I;
64            if(INRANGE(I))
65            {
66
```

```
67              int in = to_Ii(I);
68              if(xel[in].xelType==1 || xel[in].xelType==2)
69              {
70                dp = xel[in].p - mean;
71                xx += dp.x*dp.x;
72                xy += dp.x*dp.y;
73                yy += dp.y*dp.y;
74              }
75            }
76          }
77
78          //eigen value and eigen vector
79          float eVal = (xx+yy - sqrt((xx-yy)*(xx-yy) + 4*xy*xy))/2.0;
80          float eVal2 = (xx+yy + sqrt((xx-yy)*(xx-yy) + 4*xy*xy))/2.0;
81
82          dp = mean - xel[i].p;
83
84          xelF.set( (xx - eVal2)*dp.x + xy*dp.y, xy*dp.x + (yy - eVal2)*dp.y);
85          xelF = xelF/(2*(eVal - eVal2));
86
87          imgF.set( (xx - eVal2)*imgF.x + xy*imgF.y, xy*imgF.x + (yy - eVal2)*imgF.y);
88          imgF = imgF/(eVal - eVal2);
89
90        }
91
92        TXY<float> newPoint = xel[i].p + xelF + imgF;
93        xel[i].departingPoint = newPoint;
94        xel[i].departingXel = getXelIndex(newPoint);  //the xel in which the point is
              departing
95
96
97        if(INRANGE(xel[i].departingXel))
98        {
99          int i2 = to_Ii(xel[i].departingXel);
100
101         if(xel[i2].xelType==0 || xel[i2].xelType==2)  //point enters into xel only if
                the xel is empty or mobile
102         {
103           xel[i2].enteringPoint = xel[i2].enteringPoint + newPoint;
104           xel[i2].enteringPointCount++;
105         }
106       }
107     }
108   }
109 }
```

Listing 10: The function `bufferNewPoints()`

The variable `I` in this function, introduced in line 14 is distinct from the static `I` in line 15, which with 16 sets up the xel's references to neighbours.

Lines 22–35 combine counting the occupied neighbours with finding their mean. Lines 38–51 find the motion vector that would be imposed by the image forces alone, by a straightforward linearly interpolated gradient. This will later be replaced by a more direction-aware force model. Lines 54–90 similarly find the motion due to intrinsic forces, using algebra best described in a separate document. The remainder of the function combines these motions, finds the (x, y) to which the point would move, checks whether there is a xel that this (x, y) lies in (which may be the current xel, or another), and updates that xel with its arrival.

It is worth noting that the utility function `addPointToXelSet()` adds a point to a xel only if the xel is unoccupied: otherwise, an arriving point may be thought of as absorbed by the previous occupant. More subtly, neighbouring

points may move to non-neighbouring xels (Fig. 10). The gap between them could break the xelset apart, with subsets no longer connected by neighbour→neighbour→ . . . →neighbour paths. To maintain connectivity we refill xels that would do this, using the next function.

4.8.2 The function `interpolatePoints()`

```
void interpolatePoints(TXel2D *xel)
{
  FOR(i, Np)
  {

    TXY<float> p, p1, p2, dp;
    float dsMin = 0.02;
    int div;
    bool ifXelMoved1, ifXelMoved2;  //flags if the neighbors have moved out of their
        xels or not

    if(xel[i].xelType==0)
    {
      continue;
    }
    else
    if(xel[i].xelType==1)
    {
      p1 = xel[i].p;
      ifXelMoved1 = false;
    }
    else
    if(xel[i].xelType==2)
    {
      p1 = xel[i].departingPoint;
      ifXelMoved1 = true;
    }

    TXY<int> I2(0, 0);
    int zMod2 = xel[i].I.y % 2;
    int dj = zMod2*6;

    //iterating for each neighbour
    FOR(j, 6)
    {
      I2 = nbrI[j+dj] + xel[i].I;

      if(INRANGE(I2))
      {
        int i2 = to_Ii(I2);

        if(xel[i2].xelType==0)
        {
          continue;
        }
        else
        if(xel[i2].xelType==1)
        {
          p2 = xel[i2].p;
          ifXelMoved2 = false;
        }
```

```
51          else
52          if(xel[i2].xelType==2)
53          {
54            p2 = xel[i2].departingPoint;
55            ifXelMoved2 = true;
56          }
57
58          if(ifXelMoved1 || ifXelMoved2)
59          {
60            dp = p2 - p1;
61            float ds = dp.length();
62            div = floor(ds/dsMin);
63            dp = dp/div;
64            p = p1 + dp;
65
66            FOR(k,div-1)
67            {
68              TXY<int> I3 = getXelIndex(p);
69
70              if(INRANGE(I3))
71              {
72                int i3 = to_Ii(I3);
73                if(xel[i3].xelType!=1 && xel[i3].enteringPointCount==0)
74                {
75                  xel[i3].enteringPoint = p;
76                  xel[i3].enteringPointCount++;
77                }
78              }
79
80              p = p + dp;
81            }
82
83          }
84        }
85      }
86    }
87  }
```

Listing 11: The function `interpolatePoints()`.

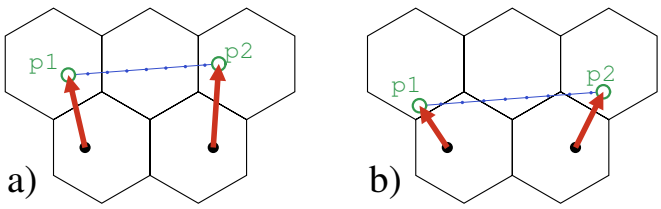


Figure 10: Two cases of neighbouring points moving to non-neighbours.

The function `interpolatePoints()` iterates over all xels to preserve the connectivity of the xelset, though not its full topology: a bubble may shrink to a blob with no hole, but it must not burst.

The code in Listing 11 first checks for each xel χ = `xel[i]` (loop: lines 3–86), whether it has become empty. If it was empty before (line 11) it has not, so no action is needed. If it is static (line 16) we set its new (x, y) position `p1` equal to the old one χ`.p` (not using the moved position computed in the force loop), and flag the xel as unaffected by movement. If it is mobile and occupied before this step (line 22), we use the moved position as `p1`, and flag χ as having a moved point. Line 28 sets up a 'clock hand' `int` vector that will point to each of six neighbours in turn

in the loop 33–85, and the next two use the parity of the row to set whether its sequence of values will start with `nbrI[0]` or `nbrI[6]`.

Line 37 uses a test defined in `xel2d.h` for whether the centre of the current neighbour χ_j would be out of the allowed rectangle in the plane, that the hexagons (over-)cover. If so, we ignore this non-existent χ_j, and move to the next neighbour. If not, we proceed as follows.

If χ_j was empty, skip it: if static, set the dummy point `p2` to its point χ_j`.p`.

If iχ_j has a mobile point (line 52) set `p2` to the moved position of the χ_j point.

We now have a pair of points `p1` and `p2`, which were neighbours in the previous state. If they have become non-neighbours, which is possible if (line 58) at least one has moved, we need to interpolate between them. In Fig. 10a, there is only one xel between `p1` and `p2`: in Fig. 10b, the line between them crosses three now-empty xels. The logic of finding every boundary crossed, to add midway points between successive crossings, would become fairly complex: but there is a simpler way. In the loop 66–81 we find points `p` at a small spacing `ds` along the segment between `p1` and `p2`. For each `p` we find `getXelIndex(p)` the xel that `p` lies in, and check whether a point has been added to that xel. If not, we assign `p` to it as its 'entering point', perhaps (harmlessly) over-writing the previous arrival.

This finite sampling is not guaranteed to add a point to every xel crossed by the segment from `p1` to `p2`: in Fig. 10b the now-empty xel at bottom right is missed. But this can only happen if the segment crosses a xel near a vertex, and a small-enough spacing `ds` then ensures that each of the other two xels that meet at that vertex *does* receive a point. Since these xels are neighbours, a chain of neighbours still connects `p1` and `p2`, and our interpolation has still achieved its purpose.

4.9 Network Extraction

The red points in Fig. 11a show a simple straight line to which a xelset shrinks, with fixed yellow ends. Branch points also occur easily (Figs. 6 and 8), but consider the straight (or with image forces, smooth) regions first.

The natural order of the red points is visually obvious, but finding it combinatorially is complicated by the way that many points on the left have four neighbours in the set. The *nearest* neighbours of each such point are in xels touching its own from above or below, but there are also neighbours in xels that touch from left or right. If all red points were like those on the right, with two neighbours each, we could simply follow the line by a rule

"After the current point p_j, added after p_{j+1} add the neighbour of p_j that is not p_j." ♣

That would work well in some places, but in others there could be up to three such neighbours.

We could implement something like "look for the nearest point in approximately the right direction", but this would involve some complicated comparisons. Vivek Na pointed out a much neater approach, which is to simplify the set, as follows.

1. For every xel point p, decide whether it is a member of a triad T of mutually neighbouring points.
2. If 1) is True, and p has no neighbours other than the other two members of T, discard p.
3. If 1) is True, and *all* the neighbours of p are all neighbours of the other two members of T, discard p.

The reasoning is that a triad of points potentially contains a xel point that is not relevant to the topology of the network. If points p, q and r are mutual neighbours, and p can only be directly reached from s and t, which are both also neighbours of q and r, then any points connected via p are also connected to q and r — which are connected. So the points remain connected even if we remove p, which thus does not separate the set.

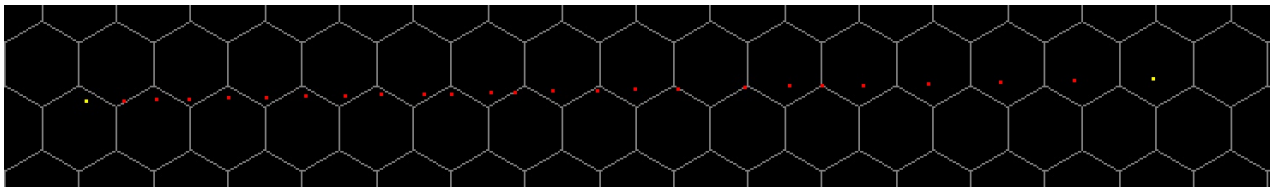
Doing this to the set in Fig. 11a produces Fig. 11b, which is far simpler.

Once the above has been done for every xel, we are left with a simplified network, where no point has four, five or six neighbours. (Proving this needs looking through a number of cases, and the assumption that the xelset cannot shrink further.)

Each xel point now belongs to one of the following cases:

- 0 neighbours : Orphan point
- 1 neighbour : End point (whisker tip)

a)



b)

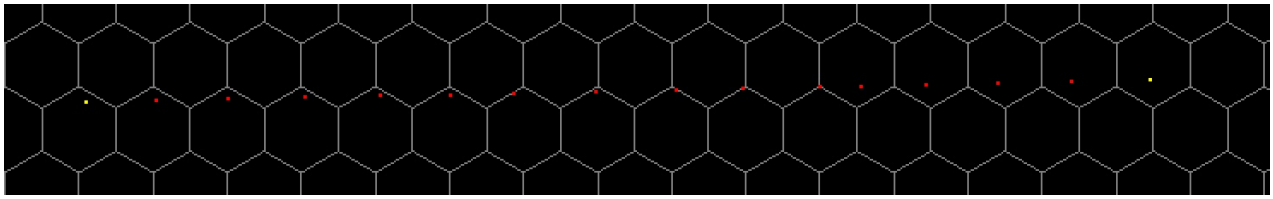


Figure 11: **(a)** A typical straight line to which a xelset shrinks with only intrinsic forces, and **(b)** a reduced version of (a).

- 2 neighbours : Chain point
- 3 neighbours : Triple point junction

Orphan points can be discarded, and ends and triple points taken as nodes of a graph. Starting from any chain point we can apply ♣ in both directions, until we reach a node.

This is implemented within `cpuRun()` by `extractNetwork()`, which takes a xelset as argument. Lines 5–120 of listing 12 below perform the cleaning: the remaining lines do the searching needed to extract the network, as described in the comments, and finally print it out.

```
1  void extractNetwork(TXel2D *xel)
2  {
3
4    //cleaning unwanted xels
5    FOR(i, Np)
6    {
7      if(xel[i].xelType!=0) //if the xel is non-empty
8      {
9
10       int dj = (xel[i].I.y % 2)*6;
11
12       TXY<int> nbrList0[6];   //nbr list of xel itself
13       TXY<int> nbrList1[6];   //nbr list of 1st nbr
14       TXY<int> nbrList2[6];   //nbr list of 2nd nbr
15       int nbrCount0 = 0;      //nbr count of xel itself
16       int nbrCount1 = 0;      //nbr count of 1st nbr
17       int nbrCount2 = 0;      //nbr count of 2nd nbr
18
19
20       //calculating nbr list of xel
21       nbrCount0 = 0;
22       FOR(j, 6)
23       {
24         //if a nbr is non-empty, it adds in nbrList0
25         TXY<int> I2 = nbrI[j+dj] + xel[i].I;
26         if(INRANGE(I2))
27           if(xel[to_Ii(I2)].xelType != 0)
28           {
29             nbrList0[nbrCount0++] = I2;
30           }
31       }
32
```

```
33        //spanning across the 6 triad formed
34        FOR(j, 6)
35        {
36          //1st nbr of triad
37          TXY<int> nbrI1 = nbrI[j+dj] + xel[i].I;
38          int nbri1 = to_Ii(nbrI1);
39
40          //2nd nbr of triad
41          TXY<int> nbrI2 = nbrI[(j+1)%6+dj] + xel[i].I;
42          int nbri2 = to_Ii(nbrI2);
43
44          if(INRANGE(nbrI1) && INRANGE(nbrI2))
45            if(xel[nbri1].xelType != 0 && xel[nbri2].xelType != 0 )
46            {
47              //we have detected three nbr-ing points forming a triad
48
49              //calculating nbr list of nbr 1
50              int dk1 = (xel[nbri1].I.y % 2)*6;
51              int dk2 = (xel[nbri2].I.y % 2)*6;
52              nbrCount1 = 0;
53              nbrCount2 = 0;
54
55              FOR(k, 6)
56              {
57                TXY<int> I2;
58                I2 = nbrI[k+dk1] + xel[nbri1].I;
59
60                if(INRANGE(I2))
61                  if(xel[to_Ii(I2)].xelType != 0)
62                  {
63                    nbrList1[nbrCount1++] = I2;
64                  }
65
66                if(INRANGE(I2))
67                  I2 = nbrI[k+dk2] + xel[nbri2].I;
68                  if(xel[to_Ii(I2)].xelType != 0)
69                  {
70                    nbrList2[nbrCount2++] = I2;
71                  }
72
73              }
74
75              //a xel to be accountable should:
76                // have nbr other than the triad
77                // and other than the nbrs of the triad
78              bool triadPassed = false;
79              FOR(l, nbrCount0)
80              {
81
82                if(nbrList0[l] == nbrI1)
83                  continue;
84
85                if(nbrList0[l] == nbrI2)
86                  continue;
87
88                bool nbrSharedByTraidMember1 = false;
89                bool nbrSharedByTraidMember2 = false;
90
```

```
91                FOR(l1, nbrCount1)
92                if(nbrList0[l] == nbrList1[l1])
93                {
94                  nbrSharedByTraidMember1 = true;
95                  break;
96                }
97                if(nbrSharedByTraidMember1)
98                  continue;
99
100               FOR(l2, nbrCount2)
101               if(nbrList0[l] == nbrList2[l2])
102               {
103                 nbrSharedByTraidMember2 = true;
104                 break;
105               }
106               if(nbrSharedByTraidMember2)
107                 continue;
108
109               triadPassed = true;
110
111             }
112
113             if(!triadPassed)
114             {
115               cleanXel(xel, xel[i].I);
116             }
117           }
118       }
119     }
120   }
121   //cleaning unwanted xel completed
122
123
124   //assigning network type number
125   FOR(i, Np)
126   {
127     if(xel[i].xelType!=0) //if the xel is non-empty
128     {
129
130       int dj = (xel[i].I.y % 2)*6;
131
132       //calculating nbr list of xel
133       int nbrCount = 0;
134       FOR(j, 6)
135       {
136         TXY<int> I2 = nbrI[j+dj] + xel[i].I;
137         if(INRANGE(I2))
138           if(xel[to_Ii(I2)].xelType != 0)
139             nbrCount++;
140       }
141
142       if(nbrCount==0)
143         xel[i].networkType = 0;
144       else
145       if(nbrCount==1)
146         xel[i].networkType = 1;
147       else
148       if(nbrCount==2)
```

```
149         xel[i].networkType = 2;
150       else
151       if(nbrCount > 2)
152         xel[i].networkType = 3;
153
154
155     }
156     else
157       xel[i].networkType = -1;
158   }
159
160
161
162   //extracting branch tree out of xel network
163
164   //step-0 clear off the old network
165   tree.nodeList.clear();
166   tree.branchList.clear();
167   tree.pendingBranchIndexList.clear();
168
169
170   //step-1 finding out the first node
171   FOR(i, Np)
172   {
173     if(xel[i].networkType == 1)
174     {
175       //adding first branch to tree
176       tree.nodeList.push_back(xel[i].I);
177
178       int dj = (xel[i].I.y % 2)*6;
179       FOR(j, 6)
180       {
181         TXY<int> I2 = nbrI[j+dj] + xel[i].I;
182         if(INRANGE(I2))
183           if(xel[to_Ii(I2)].xelType != 0)
184           {
185             TXelBranch branch;
186             branch.startI = xel[i].I;
187             branch.startNodeIndex = 0;
188             branch.pointList.push_back(xel[i].I);
189             branch.pointList.push_back(I2);
190             tree.branchList.push_back(branch);
191             tree.pendingBranchIndexList.push_back(0);
192             break;
193           }
194       }
195       break;
196     }
197   }
198
199   //step 2 finding branches
200   int ctr = 0;
201   while(tree.pendingBranchIndexList.size() > 0)
202   {
203     int bI = tree.pendingBranchIndexList[0];  //branch index
204     TXY<int> I = tree.branchList[bI].pointList.back();
205
206     int branchPointCount = tree.branchList[bI].pointList.size();
```

```
207
208     TXY<int> prevI = tree.branchList[bI].pointList[branchPointCount-2];
209
210     int i = to_Ii(I);
211     int dj = (xel[i].I.y % 2)*6;
212
213     if(xel[i].networkType == 1)
214     {
215       tree.nodeList.push_back(I);
216       tree.branchList[bI].endI = I;
217       tree.branchList[bI].endNodeIndex = tree.nodeList.size()-1;
218       tree.pendingBranchIndexList.erase(tree.pendingBranchIndexList.begin());
219       continue;
220     }
221     else
222     if(xel[i].networkType == 3)
223     {
224
225       int nodeCount = tree.nodeList.size();
226       bool oldNodeMatched = false;
227       FOR(k, nodeCount)
228       {
229         if(I == tree.nodeList[k])
230         {
231           oldNodeMatched = true;
232
233           tree.branchList[bI].endI = I;
234           tree.branchList[bI].endNodeIndex = k;
235           tree.branchList[bI].pointList.push_back(I);
236           tree.pendingBranchIndexList.erase(tree.pendingBranchIndexList.begin());
237         }
238       }
239
240       if(!oldNodeMatched)
241       {
242         tree.nodeList.push_back(I);
243         tree.branchList[bI].endI = I;
244         tree.branchList[bI].endNodeIndex = tree.nodeList.size()-1;
245         tree.pendingBranchIndexList.erase(tree.pendingBranchIndexList.begin());
246
247         FOR(j, 6)
248         {
249           TXY<int> I2 = nbrI[j+dj] + xel[i].I;
250
251           int i2 = to_Ii(I2);
252
253           if(I2 == prevI)
254             continue;
255
256           if(INRANGE(I2))
257             if(xel[i2].xelType != 0)
258             {
259               TXelBranch branch;
260               branch.startI = I;
261               branch.startNodeIndex = tree.nodeList.size()-1;
262               branch.pointList.push_back(I);
263               branch.pointList.push_back(I2);
264               tree.branchList.push_back(branch);
```

```
265                 tree.pendingBranchIndexList.push_back(tree.branchList.size()-1);
266               }
267           }
268       }
269
270       continue;
271     }
272     else
273     if(xel[i].networkType == 2)
274     {
275       FOR(j, 6)
276       {
277         TXY<int> I2 = nbrI[j+dj] + xel[i].I;
278         int i2 = to_Ii(I2);
279         if(INRANGE(I2))
280           if(xel[i2].xelType != 0)
281           {
282             //situations when we don't need to consider a nbr as an advancing chain
283             if(branchPointCount == 1)
284             {
285               int dk = (tree.branchList[bI].startI.y % 2)%6;
286               bool nbrMakesDifferentBranch = false;
287               FOR(k, 6)
288               {
289                 TXY<int> nodeNbrI = nbrI[k+dk] + tree.branchList[bI].startI;
290                 if(I2 == nodeNbrI)
291                 {
292                   nbrMakesDifferentBranch = true;
293                   break;
294                 }
295               }
296               if(nbrMakesDifferentBranch)
297               {
298                 continue; //we don't need to consider such point
299               }
300
301             }
302
303             if(I2 == prevI)
304               continue;
305
306             //////////// now the point is to be added
307             if(xel[i2].networkType == 2)
308             {
309               //I2.print();
310               tree.branchList[bI].pointList.push_back(I2);
311             }
312             else
313             if(xel[i2].networkType == 1)
314             {
315               //I2.print();
316               tree.nodeList.push_back(I2);
317               tree.branchList[bI].endI = I2;
318               tree.branchList[bI].endNodeIndex = tree.nodeList.size()-1;
319               tree.branchList[bI].pointList.push_back(I2);
320               tree.pendingBranchIndexList.erase(tree.pendingBranchIndexList.begin());
321             }
322             else
```

```
323             if(xel[i2].networkType == 3)
324             {
325               int nodeCount = tree.nodeList.size();
326               bool oldNodeMatched = false;
327               FOR(k, nodeCount)
328               {
329                 if(I2 == tree.nodeList[k])
330                 {
331                   oldNodeMatched = true;
332
333                   tree.branchList[bI].endI = I2;
334                   tree.branchList[bI].endNodeIndex = k;
335                   tree.branchList[bI].pointList.push_back(I2);
336                   tree.pendingBranchIndexList.erase(tree.pendingBranchIndexList.begin())
                        ;
337
338                 }
339               }
340
341               if(!oldNodeMatched)
342               {
343                 tree.nodeList.push_back(I2);
344                 tree.branchList[bI].endI = I2;
345                 tree.branchList[bI].endNodeIndex = tree.nodeList.size()-1;
346                 tree.branchList[bI].pointList.push_back(I2);
347                 tree.pendingBranchIndexList.erase(tree.pendingBranchIndexList.begin());
348
349                 int dk = (I2.y % 2)*6;
350                 FOR(k, 6)
351                 {
352                   TXY<int> I3 = nbrI[k+dk] + I2;
353
354                   if(I3 == I)
355                     continue;
356
357                   int i3 = to_Ii(I3);
358
359                   if(INRANGE(I3))
360                     if(xel[i3].xelType != 0)
361                     {
362                       TXelBranch branch;
363                       branch.startI = I2;
364                       branch.startNodeIndex = tree.nodeList.size()-1;
365                       branch.pointList.push_back(I2);
366                       branch.pointList.push_back(I3);
367                       tree.branchList.push_back(branch);
368                       tree.pendingBranchIndexList.push_back(tree.branchList.size()-1);
369                     }
370                 }
371               }
372             }
373           }
374       }
375
376     }
377   }
378
379   //cutting off repeated branches
```

```
380   int branchCount = tree.branchList.size();
381   bool *ifBranchRemovable = new bool[branchCount];
382   FOR(i, branchCount)
383   {
384     ifBranchRemovable[i] = false;
385     FOR(j, i)
386     {
387       if( (tree.branchList[i].startI == tree.branchList[j].endI) && (tree.branchList[i].
              endI == tree.branchList[j].startI))
388       {
389         ifBranchRemovable[i] = true;
390         break;
391       }
392     }
393   }
394
395   for(int i=branchCount-1;i>=0;i--)
396   {
397     if(ifBranchRemovable[i])
398       tree.branchList.erase(tree.branchList.begin() + i);
399   }
400
401
402   tree.print();
403 }
```

Listing 12: The function `extractNetwork()`.

4.9.1 Removing Crosslinks

There are several unwanted cross-links visible in Fig. 5b, but these do not appear in the extracted network Fig. 8. This is because we eliminate branches that join nodes that may be bright, but travel mostly through darkness. (One undesirable segment remains at upper right: if what should be a whisker end joins to such a segment, they do not form a recognised node.)

The function `removeCrossLinks()` simply looks at each branch chain in turn, ignoring any with fewer than two points between the nodes that are its ends. For each longer branch, it uses bilinear interpolation to find a great level at each point further than one step from the ends. If the mean of these values is less than 0.1, the branch is discarded.

Evidently, for applications, the discard threshold 0.1 should be replaced by an adjustable parameter. Moreover, this removal criterion would probably work better with a less-blurred version of the original input than is best for the first convergence. It should be integrated with a cycle of initial high blur (and lower resolution, for economy), followed by a tighter fit to less blurred images.

4.9.2 Simplifying branch points

The returned network often branches from a triad of points, rather than from one point. It would be straightforward to replace the triad with its centre, tidying the network and branching at a single point, but this has not yet been implemented.

It may be best done in combination with quantitative measures (for instance, of branch thickness), for the best estimates of branching angles and other quantities of medical interest.