Xelular Sets Gradient

Tim Poston

April 22, 2014

1 Scalar intensity gradient

Consider this fragment of the function `bufferNewPoints(TXel2D *xel)`, lines 232–246 of the current[1] `xel2d.cpp`.

```
1        //calculating image force
2        if(nbrCount < 6)
3        {
4          TXY<float> _p = xel[i].p/pitchX;
5          int X1 = floor(_p.x);
6          int Y1 = floor(_p.y);
7          if(X1>=0 && X1<imageWidth-1 && Y1>=0 && Y1<=imageHeight-1)
8          {
9            float dy = _p.y - Y1;
10           float dx = _p.x - X1;
11           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]);
12           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]);
13           imgF = imgF * imageForceScale;
14         }
15       }
```

Listing 1: Computing image force

What is it doing, why is it doing it, and can we make it less costly? In Fig. 1 we have a point $p = (x, y)$

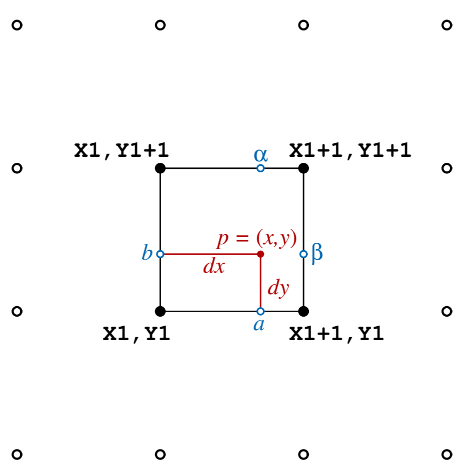

Figure 1: A point p off the grid of pixel centres, and the `dx` and `dy` from the sides found in lines 5 and 6 of the above code.

between four pixel centre points, taking the pixels as a square grid with unit steps. Earlier steps have found (x, y) as the current position of a xel point, and we want to push it so as to increase the value of the estimated grey level at that point (x, y). We only have *specified* grey levels `imgY` at the grid points, so for clarity let $\mathcal{I}(x, y)$ denote the estimated values that vary continuously, between them. Along the edges of a square like the one in

[1] as of 16 April 2014

Fig. 1 we estimate by linear interpolation, so that the estimated values of $\mathcal{I}$ at the points marked are

$$\begin{aligned}
\mathcal{I}(a) &= \Big(\texttt{imgY}\Big(\texttt{X1,Y1}\Big)\Big) + \Bigg(\Big(\texttt{imgY}\Big(\texttt{X1+1,Y1}\Big)\Big) - \Big(\Big(\texttt{imgY}\Big(\texttt{X1,Y1}\Big)\Big)\Big)\Bigg)\,\texttt{dx} \\
\mathcal{I}(b) &= \Big(\texttt{imgY}\Big(\texttt{X1,Y1}\Big)\Big) + \Bigg(\Big(\texttt{imgY}\Big(\texttt{X1,Y1+1}\Big)\Big) - \Big(\Big(\texttt{imgY}\Big(\texttt{X1,Y1}\Big)\Big)\Big)\Bigg)\,\texttt{dy} \\
\mathcal{I}(\alpha) &= \Big(\texttt{imgY}\Big(\texttt{X1,Y1+1}\Big)\Big) + \Bigg(\Big(\texttt{imgY}\Big(\texttt{X1+1,Y1+1}\Big)\Big) - \Big(\Big(\texttt{imgY}\Big(\texttt{X1,Y1+1}\Big)\Big)\Big)\Bigg)\,\texttt{dx} \\
\mathcal{I}(\beta) &= \Big(\texttt{imgY}\Big(\texttt{X1+1,Y1}\Big)\Big) + \Bigg(\Big(\texttt{imgY}\Big(\texttt{X1+1,Y1+1}\Big)\Big) - \Big(\Big(\texttt{imgY}\Big(\texttt{X1+1,Y1}\Big)\Big)\Big)\Bigg)\,\texttt{dy}
\end{aligned} \tag{1}$$

expressed in the 'constant term + slope term' style. The code above writes this a little differently: apart from treating the image as a 1-dimensional array and doing explicitly what is often left to hidden pointer arithmetic (so that it writes `imgY[Y1*imageWidth+X1]` instead of, say, `img[X1][Y1]` and so on), each line of the code does the pointer arithmetic twice, instead of the three times suggested in each line of (1).

We revisit this in a moment, but first, complete describing bilinear interpolation. At an interior point p we can linearly interpolate horizontally or vertically,

$$\mathcal{I}(p) = (1-\texttt{dx})\,\mathcal{I}(b) + \texttt{dx}\,\mathcal{I}(\beta) \tag{2}$$

$$\text{or} \quad = (1-\texttt{dy})\,\mathcal{I}(a) + \texttt{dy}\,\mathcal{I}(\alpha) \tag{3}$$

giving the same answer. This particular function, however, seeks not the *value* of $\mathcal{I}$ at p, but its *gradient* there. Within the square, differentiating with respect to `dx` or `dy` is the same as differentiating with respect to x or y respectively, so from (2) and (3) we see that the slopes of $\mathcal{I}$ at p in these two directions are simply $\mathcal{I}(\beta) - \mathcal{I}(b)$ and $\mathcal{I}(\alpha) - \mathcal{I}(a)$. A look at (1) shows that these are just linear interpolations between the slopes along the edges, and that these slopes (for a unit square grid) are simply the differences along the edges.

Consequently, if we find at every[2] point $\Big(\texttt{X1,Y1}\Big)$ we find the two values

$$\texttt{dxI}\Big(\texttt{X1,Y1}\Big) = \texttt{imgY(X1+1,Y1)} - \texttt{imgY(X1,Y1)} \tag{4}$$

$$\texttt{dyI}\Big(\texttt{X1,Y1}\Big) = \texttt{imgY(X1,Y1+1)} - \texttt{imgY(X1,Y1)} \tag{5}$$

we have slopes that we can interpolate at any $p = (x, y)$ in the square:

$$\texttt{imgF.x} = (1-\texttt{dy})\,\Big(\texttt{dxI[X1][Y1]}\Big) + \texttt{dy}\,\Big(\texttt{dxI[X1][Y1+1]}\Big) \tag{6}$$

$$\texttt{imgF.y} = (1-\texttt{dx})\,\Big(\texttt{dyI[X1][Y1]}\Big) + \texttt{dx}\,\Big(\texttt{dyI[X1+1][Y1]}\Big) \tag{7}$$

This involves just four addition/subtractions, against the eight in the code above (not counting the reference arithmetic, which should anyway be replaced by pointer advancement). It requires previous creation of the arrays `dxI` and `dyI`, once, but this gradient calculation is performed many many times — whenever a point moves, at any stage of the convergence — so the change makes sense.

It is also a useful warming up exercise, for changing the code to take account of a *directional* image like the one we get from the ridge detection code.

2 Vector field effects

The output of the bottom-up vessel detector is a *vector* quantity: let us call it here, $\mathbf{v} = (v_x, v_y)$. We want to smooth and blur it (which we can do just by applying the same blurring to each of v_x and v_y), for the same

[2] except at the right-most and top of the image, where they are undefined.

reasons as before. But we also want to move points to where the strength $\|\mathbf{v}\|$ is greater, but that would waste the directionality. We want to take into account the agreement (or not) between the geometric directionality of the xelset around point p, and the image direction revealed by $\mathbf{v}$.

First, let us revisit the xelular logic that pulls the set toward a thin curve (in the absence of image forces, toward a line), as we now want the xelular directionality to interact with the image directionality. This logic is also currently in the function `bufferNewPoints(TXel2D *xel)`, along with the image logic.

2.1 Xelular force

It is worth noticing that the process of finding the xelular force includes a neighbour count: if there are neighbours on all six sides, we will not move the current point. We are not concerned to make the interior of a blob more even, any more than we need points evenly spaced along a curve. The condition `if(nbrCount < 6)` in line 2 of Listing 1 prevents the current point from moving in this round.

```
1     if(nbrCount>1 && nbrCount<6)
2     {
3
4       TXY<float> mean(0, 0);     // initialise centroid
5       int nbrCount = 0;         // and neighbour count
6
7       //calculating mean of neighbours
8       FOR(j,6)
9       {
10        I = nbrI[dj+j] + xel[i].I;   // find 2D-array index of jth neighbour xel
11        if(INRANGE(I))               // if not beyond boundary
12        {
13          int in = to_Ii(I);         // change to 1D-array index
14          if(xel[in].xelType==1 || xel[in].xelType==2)
15          {                             // if xel has a point
16            mean = mean + xel[in].p;
17            nbrCount++;
18          }
19        }
20      }
21      mean = mean/nbrCount;    // centroid now known
22
23  //   ...  in current code, image force calculations here
24
25      //initialise covariance matrix
26      float xx=0, xy=0, yy=0;
27      TXY<float> dp;
28
29      FOR(j,6)
30        {
31        I = nbrI[dj+j] + xel[i].I;   // find 2D-array index of jth neighbour xel
32        if(INRANGE(I))               // if not beyond boundary
33        {
34          int in = to_Ii(I);         // change to 1D-array index
35          if(xel[in].xelType==1 || xel[in].xelType==2)
36          {                             // if xel has a point p
37            dp = xel[in].p - mean;   // vector from mean to p
38            xx += dp.x*dp.x;        // contribution to
39            xy += dp.x*dp.y;       // covariance
40            yy += dp.y*dp.y;      // matrix
41          }
42        }
43
44      //eigenvalue and eigenvector
45      float tr = xx + yy ;    // trace of matrix
46      float tr2 = tr/2.0 ;    // half trace of matrix
```

```
      float di = (xx-yy)^2 + 4*xy^2 ;  // discriminant of eigen-equation
      float disq = sqrt(di) / 2.0 ;
      float eVal = tr - disq ;     // small eigenvalue > 0
      float eVal2 = tr + disq;    // large eigenvalue > 0

      dp = mean - xel[i].p;    // vector from point p to mean of neighbours

      // projection matrix (up to a scalar) onto L-perp
      float m11 =  xx - eVal2 ;
      float m12 =  xy           ;
      float m22 =  yy - eVal2 ;
      xelF.set( m11*dp.x + m12*dp.y,  m12*dp.x + m22*dp.y);
      xelF = xelF/(2*(eVal - eVal2));   // should be xelF/eval2, as in math below?
              // factor 2 is to prevent overshooting, should be a tuned parameter.
                                   //  Tests needed

      // important use of matrix M to restrict gradient to sideways direction
      imgF.set( m11*imgF.x + m11*imgF.y,     m11*imgF.x + m22*imgF.y);
      imgF = imgF/disq;   // pure projection makes more sense here
        // note that lambda1-lambda2 == disq
                // trace would be more natural?   Test
      }
```

Listing 2: Computing xelular force

Let us see the mathematics behind Listing 1.

To smooth a curve C, we move a point P and its neighbours (points in neighbouring xels) toward their best-fit line L. We do not move it *along* the line L: the even spacing this works toward is often incompatible with 'one point in each xel that C crosses' — if it persists when shrunk to a line.

2.1.1 Finding a line

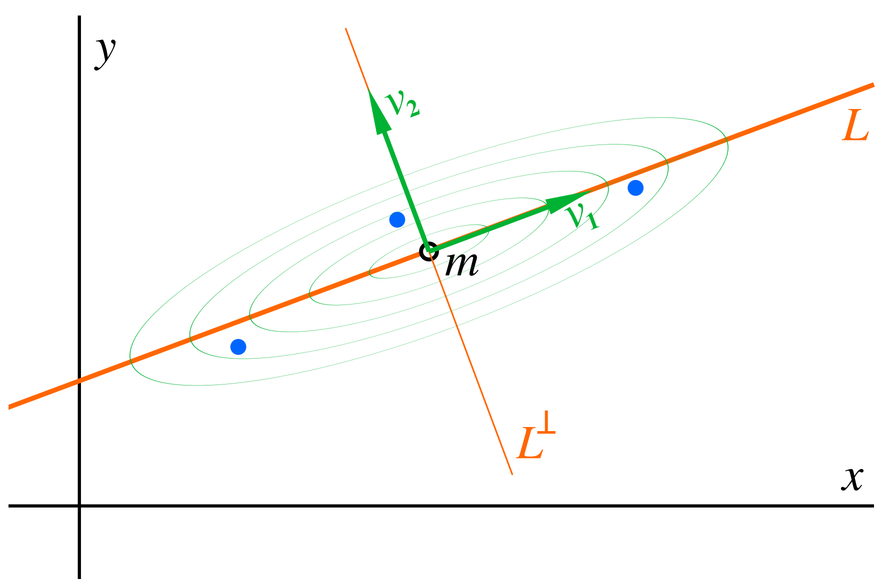

Figure 2: Fitting a line to dots.

Ignoring xels for now, suppose we have some points $P_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ and we want to know the line L they best fit: the end game is to push particular points toward L, rather than find L itself, as image forces will gently bend the results. But, conceptually, think first about L.

Suppose and the line L has the equation $ax + by = c$. How do do we choose it? First, find the centroid

$$m \;=\; (\overline{x}, \overline{y}) \;=\; \frac{1}{n}\sum_{i=1}^{n}(x_i, y_i) \;=\; \left(\frac{\sum_{i=1}^{n} x_i}{n}, \frac{\sum_{i=1}^{n} y_i}{n}\right) \tag{8}$$

as done in line 21 of Listing 66 above, giving the 2D point `mean = (xx,yy)` in the code. Set

$$\widetilde{x}_i = x_i - \overline{x}\,, \quad \widetilde{y}_i = y_i - \overline{x}\,, \tag{9}$$

Then the loop 29–42 finds the matrix

$$\mathbf{T} \;=\; \begin{bmatrix} A & B \\ B & C \end{bmatrix} \;=\; \sum_{i=1}^{n}\begin{bmatrix} \widetilde{x}_i^2 & \widetilde{x}_i\widetilde{y}_i \\ \widetilde{x}_i\widetilde{y}_i & \widetilde{y}_i^2 \end{bmatrix}. \tag{10}$$

Each individual matrix summand

$$\mathbf{T}_i \;=\; \begin{bmatrix} \widetilde{x}_i^2 & \widetilde{x}_i\widetilde{y}_i \\ \widetilde{x}_i\widetilde{y}_i & \widetilde{y}_i^2 \end{bmatrix} \tag{11}$$

takes all vectors to the line through $\mathbf{0}$ and $\mathbf{v}_i = (\widetilde{x}_i, \widetilde{y}_i)$, as each column is a multiple of $\mathbf{v}_i$. The vector $\mathbf{v}_i$ itself maps to $\|\mathbf{v}_i\|^2\mathbf{v}_i$ and any vector $\mathbf{u}$ orthogonal to $(\widetilde{x}_i, \widetilde{y}_i)$ has $\mathbf{T}_i\mathbf{u} = 0$, giving eigenvalues $\|\mathbf{v}_i\|^2$ and 0. (For instance, if $\mathbf{v}_i = \left[\begin{smallmatrix}1\\1\end{smallmatrix}\right]$ then $\mathbf{M}_i = \left[\begin{smallmatrix}1&1\\1&1\end{smallmatrix}\right]$ which maps $\mathbf{v}_i$ to $\left[\begin{smallmatrix}2\\2\end{smallmatrix}\right] = 2\mathbf{v}_i$, and $\left[\begin{smallmatrix}-1\\1\end{smallmatrix}\right]$ to $\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$.)

Add all these projections and we have $\mathbf{T}$, a matrix which (being symmetric) can be equated to

$$\mathbf{T} \;=\; \lambda_1\mathbf{P}_1 + \lambda_2\mathbf{P}_2 \tag{12}$$

where $\lambda_1 \geq \lambda_2 \geq 0$ are the eigenvalues of $\mathbf{T}$, $\mathbf{P}_1$ is orthogonal projection in the direction of $\mathbf{v}_2$ onto the line $L = \{s\,\mathbf{v}_1 \mid s \in \mathbb{R}\}$, and $\mathbf{P}_2$ is orthogonal projection onto $L^{\perp}$, at right angles to it.

2.1.2 Projecting to the line

The code does not need to find L explicitly: what we need are the projection operations.

So as not to push points in the direction along L, belonging to the big eigenvector $\lambda_1 \;=\;$ `eVal2`, we can use

$$\begin{bmatrix} A-\lambda_1 & B \\ B & C-\lambda_1 \end{bmatrix} = (\lambda_1-\lambda_1)\mathbf{P}_1 + (\lambda_2-\lambda_1)\mathbf{P}_2 = (\lambda_2-\lambda_1)\mathbf{P}_2 \tag{13}$$

and divide it by λ_1. When we apply

$$\mathbf{M} \;=\; \frac{1}{\lambda_1}\begin{bmatrix} A-\lambda_1 & B \\ B & C-\lambda_1 \end{bmatrix} = \frac{\lambda_2-\lambda_1}{\lambda_1}\mathbf{P}_2 \tag{14}$$

$$\text{or} \qquad \mathbf{P}_2 \;=\; \frac{1}{\lambda_1-\lambda_2}\begin{bmatrix} A-\lambda_1 & B \\ B & C-\lambda_1 \end{bmatrix} \tag{15}$$

to a general vector $\mathbf{u}$, we get a vector pointing from $\mathbf{u}$ to the line L, at right angles.

Using $\mathbf{P}_2$ — as the current code does, in line 59 of the version above — projects with a 'force' independent of how tightly aligned the points already are, and may be unstable when the eigenvalues approach equality (and the absence of a direction of alignment), so it may be best to use $\mathbf{M}$, though the current code seems to work. Experiment needed!

If the eigenvalues are almost equal, the vector $\mathbf{Mu}$ is very small. If they are very different (*i.e.*, for points almost in line) then $l = (\lambda_2 - \lambda_1)/\lambda_1$ is near -1, and adding $\mathbf{Mu}$ to the vector $\mathbf{u}$ takes it almost to L. But since $\lambda_1 \geq \lambda_2 \geq 0$, we always have $-1 \leq l \leq 0$, and $\mathbf{Mu}$ is a step toward L.

The lengthwise strength of the local geometry is clearly in the direction of L, to be represented by an eigenvector belonging to the large eigenvalue λ_1. Two such eigenvectors are the columns of

$$\begin{bmatrix} A-\lambda_2 & B \\ B & C-\lambda_2 \end{bmatrix} \tag{16}$$

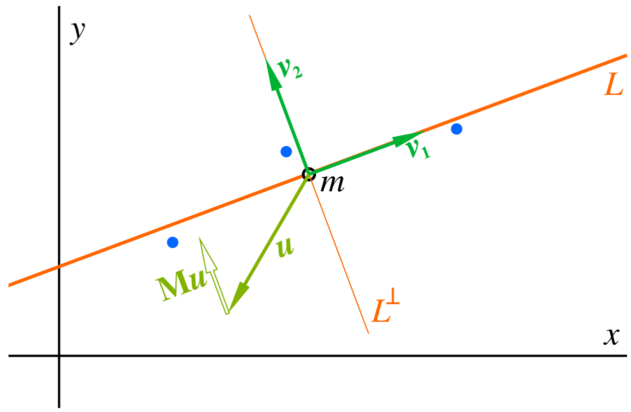

Figure 3: Moving toward the line.

but we want the *size* to become small as λ_1 and λ_2 approach each other, and we don't want it to increase with A, B and C (those are bigger just with more neighbours). Dividing by λ_2 would explode as that goes to zero (the set approaching a straight line) so we want a directionality vector with size $(\lambda_1 - \lambda_2)/\lambda_1$: for instance, normalise a column of (16) and multiply it by that number.

Call this vector 'fineness' (how much like fine thread, in what direction) and denote it by $\mathbf{f} = (f_x,\ f_y)$.

2.2 Vector intensity gradient

It is easy to differentiate $\mathbf{v} = (v_x,\ v_y)$ with respect to x and y: just do to each of v_x and v_y what Listing 1 does to `imgY`. But instead of one vector `imgF.x` this gives two vectors ($\mathbf{v}^x$ and $\mathbf{v}^y$, say) so how should point p move?

We could say that if the xelset already looks horizontal, with $\mathbf{f} = (1,\ 0)$, say, we move entirely according to $\mathbf{v}^x$, ignoring $\mathbf{v}^y$, and vice versa: in general, move according to $f_x\mathbf{v}^x + f_y\mathbf{v}^y$. But this would go badly, because v_x is only 'the intensity of horizontal fineness' up to sign. (We cannot insist that both v_x and v_y are always positive, because that would allow only / directions, not \ ones.) If it happens to have a negative sign, then the gradient vector points in the steepest direction of *decreasing* absolute intensity, not the way we want to go.

So, to suppress this ambiguity, let us treat $\mathbf{v}$ as a complex number $v_x + iv_y$ and take its complex square,

$$V_x + iV_y \ = \ \mathbf{V} \ = \ (v_x + iv_y)^2 \ = \ \left(v_x^2 - v_y^2\right) + i\,(2v_xv_y)\,. \tag{17}$$

This is positive real if $\mathbf{v}$ is horizontal (in either direction), negative real if $\mathbf{v}$ is vertical, and so on. For an unoriented / direction it is around $+i$, for a \ it is $-i$. Similarly, set

$$F_x + iF_y \ = \ \mathbf{F} \ = \ (f_x + if_y)^2 \ = \ \left(f_x^2 - f_y^2\right) + i\,(2f_xf_y)\,. \tag{18}$$

as a $\pm$-free quantity to describe the fineness of the xelset.

Differentiating V_x and V_y gives steepest-direction vectors $\mathbf{V}^x$ and $\mathbf{V}^y$ for the horizontal and vertical aspects respectively of $\mathbf{V}$, toward 0 when V_x or V_y are negative, away from it when they are positive. So with similar sign behaviour for $\mathbf{F}$, the direction of the mixture

$$F_x\mathbf{V}^x + F_y\mathbf{V}^y \tag{19}$$

looks right to move the point p. Note that if $\mathbf{F} = (-1, 0)$, as for a vertical thread, (19) will push our point p in a way that *decreases* the horizontal strength of $\mathbf{v}$. Horizontal data will repel vertical curves, and vice versa.

The length may want a little tweaking, as the squaring in (17) and (18) means that doubling $\mathbf{v}$ quadruples $\mathbf{V}$ and its derivatives (and hence (19)), and doubling $\mathbf{f}$ does the same for $\mathbf{F}$. We may do better with a more linear response.