Sparse Field Methods - Technical Report

Shawn Lankton

July 6, 2009

Abstract

Active contour methods for image segmentation allow a contour to deform iteratively to partition an image into regions. Active contours are often implemented with level set methods because of their power and versatility. The primary drawback of level set methods is that they are slow to compute. This document describes the sparse field method (SFM) proposed by Whitaker that allows one to implement level set active contours efficiently. The algorithm is described in detail, specific notes are given about implementation, and the reader is referred to resources including source codes.

1 Introduction

Active contour methods for image segmentation allow a contour to deform iteratively so as to partition an image into regions corresponding to the scene represented by the image [3]. There are various ways to represent a contour. Level set methods [6, 5] are used quite often because they are simple to implement, and allow very complex curve behavior.

In the level set framework, the contour C is embedded in a higher-dimensional function. For example a simple curve on a 2D plane can be embedded in a 3D surface ϕ. By convention, C is represented as the zero-level-set of ϕ such that the curve is located where ϕ crosses a plane at the 0 level. Thus, on the interior of C, $\phi < 0$ and on the exterior of C, $\phi > 0$. Typically, ϕ is chosen to be the signed distance function (SDF) such that the $\|\phi(x)\|$ is equal to the distance from x to the closest point on C and interior points have a negative sign. This choice is made so that first and second derivatives can be taken about the curve with ease. As a simple example of this representation, imagine that C is a circle. The corresponding SDF, ϕ would be an inverted cone whose intersection with the zero plane exactly matches C as shown in Figure 1.

The primary drawback of level set methods is that they are slow to compute. In their truest form, many computations are required to maintain ϕ as the contour changes. The key to reducing complexity is to notice that only the area of ϕ where $\phi(x) \approx 0$ is important to accurately represent the curve. This has led to proposals of various *narrow-band* algorithms that reduce the computational complexity by only performing

Figure 1: The contour C and its corresponding SDF, ϕ.

calculations near the zero level set. The sparse field method (SFM) proposed by Whitaker in [8] remains one of the most efficient algorithms to maintain a minimal, but accurate representation of ϕ. Using the SFM can drastically reduce computation times for level set methods.

Whitaker's original paper outlines his algorithm and presents evidence that his method requires fewer computations and produces less error than other narrow-band techniques. However, in [8] the algorithm is described briefly and in abstract terms. The remainder of this document will describe the SFM in detail and make specific comments regarding implementation and some natural extensions. The algorithm and its limitations are discussed, and finally, the reader is directed toward useful resources including source code.

2 Sparse Field Algorithm

The sparse field method (SFM) uses lists of points that represent the zero level set as well as points adjacent to the zero level set. By using these lists and carefully moving points to and from the appropriate list a very efficient representation of ϕ can be maintained.

These lists are implemented as *doubly-linked-lists*. These lists have the properties that elements can be added dynamically, and that elements can be removed from the middle of the list. This type of data-structure is available in most programming languages (for example, the `vector` class in C++).

Five lists are used in the SFM to represent five different levels:

$$
\begin{array}{lcccccl}
\mathtt{L}_0 & \rightarrow & [\text{-}0.5 & , & 0\ .5] & \rightarrow & \mathtt{Lz} \\
\mathtt{L}_{-1} & \rightarrow & [\text{-}1.5 & , & \text{-}0.5) & \rightarrow & \mathtt{Ln1} \\
\mathtt{L}_1 & \rightarrow & (\ 0.5 & , & 1\ .5] & \rightarrow & \mathtt{Lp1} \\
\mathtt{L}_{-2} & \rightarrow & [\text{-}2.5 & , & \text{-}1.5) & \rightarrow & \mathtt{Ln2} \\
\mathtt{L}_2 & \rightarrow & (\ 1.5 & , & 2\ .5] & \rightarrow & \mathtt{Lp2}
\end{array}
$$

Each list holds the x, y, and z location of pixels in the image. In addition to the lists, two arrays are used. The first is the ϕ array. This is the same size as the image domain and should be maintained at full

floating point precision. The second array is a label map the same size as ϕ. This label map is used to record the status of each point and provides a way to look up which list a point belongs to. The label map will only contain the values $\{-3, -2, -1, 0, 1, 2, 3\}$.

2.1 Initialization

Procedure 1 takes in a binary image (`init`) and returns a fully initialized arrays for the label map (`label`), ϕ (`phi`) and the five lists representing levels -2 through 2. The binary image (`init`) should have values 0 and 1 where 1's represent foreground pixels, and 0's represent background pixels. Furthermore, the notation: `N(p)` is used to denote the neighborhood of an point p. This represents the 4-neighborhood in 2D (up, down, left, right) and the 6-neighborhood in 3D (up, down, left, right, forward, and backward).

Procedure 1 Initialization

```
    // Pre-condition labelmap and phi
 1: for each point p in the domain.
 2:     if(init(p) == 0) label(p) =  3, phi(p) =  3
 3:     if(init(p) == 1) label(p) = -3, phi(p) = -3

    // Find the zero-level set
 4: for each point p in the domain.
 5:     if(init(p) == 1 AND any point in N(p) == 0)
 6:         add p to Lz
 7:         label(p) = 0, phi(p) = 0

    // Find the +1 and -1 level set
 8: for each point p in Lz
 9:     for each point q in N(p)
10:         if(label(q)==-3) add q to Ln1, label(q)=-1, phi(q)=-1
11:         if(label(q)== 3) add q to Lp1, label(q)= 1, phi(q)= 1

    // Find the +2 and -2 level set
12: for each point p in Ln1
13:     for each point q in N(p)
14:         if(label(q)==-3) add q to Ln2, label(q)=-2, phi(q)=-2

15: for each point p in Lp1
16:     for each point q in N(p)
17:     if(label(q)== 3) add q to Lp2, label(q)= 2, phi(q)= 2
```

2.2 Evolving the Contour

Once the data structures have been initialized with Procedure 1, the level sets may be deformed in order to minimize some segmentation energy. There are many such energies, and the choosing of appropriate segmentation energies for active contour segmentations is a topic of much research. See [1] for an introduction. One very popular active contour energy is the so-called Chan-Vese energy [2]

$$E = \int_{interior} (I - \mu_1)^2 + \int_{exterior} (I - \mu_2)^2, \tag{1}$$

which has the corresponding evolution equation

$$F = (I - \mu_1)^2 - (I - \mu_2)^2. \tag{2}$$

Note that F should only be computed along the zero level set and should be normalized such that $\|F\| < 0.5$ at each iteration.

Updating ϕ near the zero level set: Once F has been computed and normalized, Procedure 2 describes how to update ϕ along the zero level set as well as the four other lists of adjacent points. Note the use of five additional lists that temporarily hold points that are changing status:

S_0	=	`Sz`	$\rightarrow$ Points moving to L_0
S_{-1}	=	`Sn1`	$\rightarrow$ Points moving to L_{-1}
S_1	=	`Sp1`	$\rightarrow$ Points moving to L_1
S_{-2}	=	`Sn2`	$\rightarrow$ Points moving to L_{-2}
S_2	=	`Sp2`	$\rightarrow$ Points moving to L_2

Procedure 2 can be summarized as follows:

1. Scan through L_0 and add the corresponding value of F to the existing value of ϕ at each point.
2. Points where the new value of ϕ is outside of the range [-.5 .5] are removed from L_0 and added to S_{-1} or S_1 accordingly to denote that they will be changing status to L_{-1} or L_1 respectively.
3. L_{-1} and L_1 are scanned and ϕ values are updated so that they are exactly 1 unit from their nearest neighbor in L_0. If no L_0 neighbors exist, the point is moved to S_{-2} or S_2 respectively.
4. Points in L_{-1} and L_1 whose updated ϕ value fall outside of the specified range for L_{-1} or L_1 are moved to S_0, S_{-2}, or S_2 accordingly.
5. L_{-2} and L_2 are scanned and ϕ values are updated so that they are exactly 1 unit from their nearest neighbor in L_{-1} or L_1. If no L_{-1} or L_1 neighbors exist, the point is removed from all lists, and the value of ϕ and the label map is changed to -3 or +3 accordingly.
6. Points in L_{-2} and L_2 whose updated ϕ value is outside of the specified range are removed from L_{-2} or L_2 and either moved to S_{-1} or S_1 if their value is too low, or removed from all lists if their value is too high. Points that are removed from all lists should have their corresponding point in ϕ and the label map changed to -3 or +3 accordingly.

Procedure 2 Update Level Set Lists

```
     // Update the zero level set
 1:  for each point p in Lz
 2:      add F(p) to phi(p)
 3:      if(phi(p)> .5), remove p from Lz, add p to Sp1
 4:      if(phi(p)<-.5), remove p from Lz, add p to Sn1

     // Update -1 and +1 level sets
 5:  for each point p in Ln1
 6:       if p has no neighbors q in N(p) with label(q) == 0
 7:           remove p from Ln1, add p to Sn2
 8:       else
 9:           M = MAX of points q within N(p) and with label(q) >= 0
10:           phi(p) = M-1;
11:           if(phi(p)>= -0.5) remove p from Ln1, add p to Sz
12:           if(phi(p)<  -1.5) remove p from Ln1, add p to Sn2

13:  for each point p in Lp1
14:      if p has no neighbors q in N(p) with label(q) == 0
15:          remove p from Lp1, add p to Sp2
16:      else
17:          M = MIN of points q within N(p) and with label(q) <= 0
18:          phi(p) = M+1;
19:          if(phi(p)<=   0.5) remove p from Lp1, add p to Sz
20:          if(phi(p)>    1.5) remove p from Lp1, add p to Sp2

     // Update -2 and +2 level sets
21:  for each point p in Ln2
22:      if p has no neighbors q in N(p) with label(q) == -1
23:          remove p from Ln2, label(p)=-3, phi(p)=-3
24:      else
25:          M = MAX of points q within N(p) and with label(q) >= -1
26:          phi(p) = M-1;
27:          if(phi(p)>= -1.5) remove p from Ln2, add p to Sn1
28:          if(phi(p)<  -2.5) remove p from Ln2, label(p)=-3, phi(p)=-3

29:  for each point p in Lp2
30:      if p has no neighbors q in N(p) with label(q) ==  1
31:          remove p from Lp2, label(p)= 3, phi(p)= 3
32:      else
33:          M = MIN of points q within N(p) and with label(q) <=  1
34:          phi(p) = M+1;
35:          if(phi(p)<=  1.5) remove p from Lp2, add p to Sp1
36:          if(phi(p)>   2.5) remove p from Lp2, label(p)= 3, phi(p)= 3
```

Dealing with points that change status: Once all points in the lists L_0, L_{-1}, L_1, L_{-2}, and L_2 have been visited, it is necessary to process points that have changed status during the iteration. This is described in Procedure 3, summarized as follows:

1. Scan S_0. Move each point onto $L_0$and update the value of the label map to 0.
2. Scan S_{-1} and S_1. Move each point onto L_{-1} or L_1 accordingly and update the label map to -1 or 1. If points on L_{-1} or L_1 have neighbors with ϕ values equal to -3 or 3, add the neighbors to the appropriate S_{-2} or S_2 lists while setting their ϕ value to be 1 unit from the value of the current point.
3. Scan S_{-2} and S_2. Move each point onto L_{-2} or L_2 accordingly and update the label map to -2 or 2 depending on the sign.

By running Procedure 2 and Procedure 3 a full iteration is completed and F can be re-computed based on the new position of the contour. This process is repeated until convergence is reached.

Procedure 3 Deal with points that are changing status

```
    // Move points into zero level set.
 1: for each point p in Sz
 2:     label(p) = 0, add p to Lz, remove p from Sz

    // Move points into -1 and +1 level sets
    // and ensure -2, +2 neighbors
 3: for each point in Sn1
 4:     label(p) = -1, add p to Ln1, remove p from Sn1
 5:     for each point q in N(p)
 6:         if(phi(q)==-3) phi(q)=phi(p)-1, add q to Sn2

 7: for each point in Sp1
 8:     label(p) =  1, add p to Lp1, remove p from Sp1
 9:     for each point q in N(p)
10:         if(phi(q)== 3), phi(q)=phi(p)+1, add q to Sp2

    // Move points into -2 and +2 level sets
11: for each point p in Sn2
12:     label(p) = -2, add p to Ln2, remove p from Sn2

13: for each point p in Sp2
14:     label(p) =  2, add p to Lp2, remove p from Sp2
```

Efficiently updating statistics: When computing the movement of the contour, it is often desirable to track when a point crosses the zero level set thus changing from an *interior* point to an *exterior* point or vice versa. This can be accomplished by checking the sign of ϕ before and after adding F to the value of ϕ for points on the zero level set. (Line 2 of Procedure 2).

When a point changes sign from negative to positive it can be added to a list L_{in2out} or L_{out2in} if the sign changes from positive to negative. Then, these two lists can be processed after the iteration to update statistics more efficiently. For example, if interior and exterior means are used when computing F, it is much more efficient to alter the statistics based on the movement the few points that cross the interface than to re-compute the means inside and outside the contour at each iteration.

3 Source Code

The SFM as described here has been implemented in C++/MEX for use with MATLAB. The code will be made available at the author's website (`http://www.shawnlankton.com`) and MATLAB Central page.

4 Strengths and Limitations

The SFM has several very nice properties that make it a valuable tool for image processing. The fact that it uses lists to keep track of the points near the zero level set means that the speed of curve updates is dependent only on the length of the curve, and not the size of the image. This is very valuable when working with very large medical imaging volumes. Also, the accuracy of this method as shown in [8] means that by using the SFM, the behavior implemented energies can be well understood without concern that the specific curve implementation is affecting the outcome.

One limitation is that because forces are only computed along the zero level set, it is not possible for new curves to appear spontaneously. Instead, the initial contour may only grow, split, and merge to obtain its final shape. While this is the theoretically, "correct" behavior, many implementations make use of the ability for new contours to emerge spontaneously to fill holes and bridge gaps in image information.

5 Approximate Techniques

The SFM implements full, numerical level set methods very efficiently. However, better performance can be achieved if exact solutions are not required. In [7] a method is presented where three lists are kept that correspond to the zero, +1, and -1 level sets. Rather than maintain floating point representation of ϕ within these lists, however, this approximate method uses integer values -1, 0, and 1. With this simplification, they can achieve much faster speeds.

A further improvement was published in [4] where the number of required lists is reduced to one. A single list encodes the zero level set

making the computation of each iteration very efficient. Furthermore, in these integer-valued approximations, each part of the contour moves in whole-pixel increments. This results in slightly less accurate solutions, but allows the algorithm to converge in much fewer iterations. MATLAB Code for the algorithm in [4] can be found at: http://www.jgmalcolm.com/.

References

[1] A. Blake and M. Isard. *Active Contours*. Springer, Cambridge, 1998.

[2] T. Chan and L. Vese. Active contours without edges. *IEEE Transactions on Image Processing*, 10(2):266–277, February 2001.

[3] D. Cremers, M. Rousson, and R. Deriche. A review of statistical approaches to level set segmentation: Integrating color, texture, motion, and shape. *International Journal of Computer Vision*, 72(2):195–215, 2007.

[4] J. Malcolm, Y. Rathi, A. Yezzi, and A. Tannenbaum. Fast approximate surface evolution in arbitrary dimension. In *Proceedings of SPIE Medical Imaging*, 2008.

[5] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Cambridge University Press, New York, NY, 2003.

[6] J.A. Sethian. *Level Set Methods and Fast Marching Methods, Second Edition*. Springer, New York, NY, 1999.

[7] Y. Shi and W. Karl. A fast level set method without solving pdes. In *Proceedings of ICASSP*, volume 2, 2005.

[8] R. Whitaker. A level-set approach to 3D reconstruction from range data. *International Journal of Computer Vision*, 29(3):203–231, 1998.