# Math 671: Level set methods

# Variational level sets in shape reconstruction from unorganised data sets

Harish Narayanan

21st December, 2005

## Contents

## 1 Introductory ideas

Following in a vein similar to [Zhao et al., 2001], the goal of this project is to develop fast and accurate schemes to reconstruct shapes from unorganised data-sets using variational level set methods. The approach followed is based primarily on the gradient-flow scheme derived in class from variational considerations, and uses implementation details borrowed from [Zhao, 2004, Osher and Fedkiw, 2002]. Most notably, unlike the convection model followed in [Zhao et al., 2001], this current treatment also includes the curvature-based term (on the right hand side of Eq. (6)) neglected in their earlier work.

Given an arbitrary set $\boldsymbol{S} = \{\text{points, curves, surfaces in } N \text{ dimensions}\}$, the basic problem at hand is the determination of an $N$ dimensional surface $\Gamma$ which minimises the error defined by

$$E = \int_\Gamma d(\boldsymbol{x})ds, \tag{1}$$

where $d(\boldsymbol{x})$ is the distance function $d(\boldsymbol{x}, \boldsymbol{S})$, which is the closest distance to the data-set $\boldsymbol{S}$ from a point $\boldsymbol{x}$. Assuming $\Gamma$ to be the zero level set of a function $\varphi$, i.e.,

$$\Gamma = \{\boldsymbol{x} : \varphi(\boldsymbol{x}) = 0\}, \tag{2}$$

it is clear from level set theory that the error now becomes

$$E = \int_\Omega d(\boldsymbol{x})|\boldsymbol{\nabla}(\varphi)|\delta(\varphi)d\boldsymbol{x}, \tag{3}$$

where $\delta(\cdot)$ is Dirac's delta function.

As detailed in class (on 11/22/2005), in order to arrive at the gradient flow law, we first differentiate Eq. (3) with respect to time,

$$\dot{E} = \int_\Omega \left( d(\boldsymbol{x})|\boldsymbol{\nabla}(\varphi)|\delta^{'}(\varphi)\varphi_t + d(\boldsymbol{x})\delta(\varphi)\frac{\boldsymbol{\nabla}(\varphi)}{|\boldsymbol{\nabla}(\varphi)|} \cdot \boldsymbol{\nabla}(\varphi_t) \right) d\boldsymbol{x}. \tag{4}$$

Integrating by parts,

$$\begin{aligned} \dot{E} &= \int_\Omega \left( d(\boldsymbol{x})|\boldsymbol{\nabla}(\varphi)|\delta^{'}(\varphi) - \boldsymbol{\nabla} \cdot \left( d(\boldsymbol{x})\delta(\varphi)\frac{\boldsymbol{\nabla}(\varphi)}{|\boldsymbol{\nabla}(\varphi)|} \right) \right) \varphi_t d\boldsymbol{x} \\ &= -\int_\Omega \left( \delta(\varphi) \left( \frac{\boldsymbol{\nabla}(d) \cdot \boldsymbol{\nabla}(\varphi)}{|\boldsymbol{\nabla}(\varphi)|} + d(\boldsymbol{x})\kappa \right) \varphi_t \right) d\boldsymbol{x}, \end{aligned} \tag{5}$$

where $\kappa$ is the curvature. In order to have gradient flow, we require $\frac{\partial\varphi}{\partial t} = \frac{-\delta E}{\delta\varphi}$, where $\delta\cdot$ is a variation of $\cdot$, not the delta function. Recalling that $\frac{dE}{dt} =< \frac{\delta E}{\delta\varphi}, \varphi_t >$, we arrive at the following gradient flow law for $\varphi$.

$$\frac{\partial\varphi}{\partial t} - \boldsymbol{\nabla}(d) \cdot \boldsymbol{\nabla}(\varphi) = d(\boldsymbol{x})\kappa|\boldsymbol{\nabla}(\varphi)|, \tag{6}$$

It is Eq. (6) that we aim to solve to determine $\varphi$, and hence $\Gamma$.

## 2 Arriving at an inexpensive distance function

The first step of this solution process involves the computation of the distance function, $d(\boldsymbol{x})$. In the following, I briefly detail the methods I tried, and the algorithm used in the final implementation. Summaries of existing schemes can be found in [Mauch, 2000, Russo and Smereka, 2000], for example.

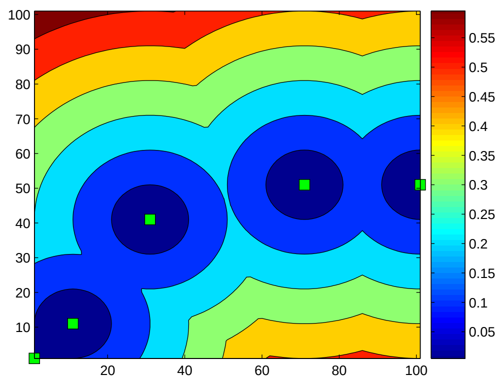
Figure 1: Distance function contours for an arbitrary set of 5 points.

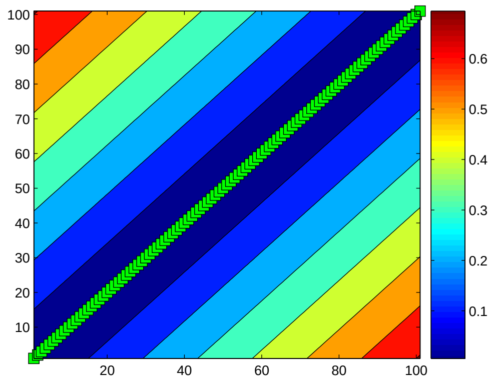
Figure 2: Distance function contours for $\boldsymbol{S}$ being the diagonal.

## 2.1 Brute force approach

As a first pass, we return to the basic definition of the distance function,

$$d(\boldsymbol{x}) = \min_{\boldsymbol{y}\in\Gamma} |\boldsymbol{x} - \boldsymbol{y}|. \tag{7}$$

In order to obtain a field over the domain $\Omega$, for every point $\boldsymbol{x}$ in $\Omega$, the algorithm determines its distance to every point in $\boldsymbol{S}$, and returns the minimum of these values. The following is the relevant MATLAB code snippet.

```
% A very simple distance function evaluator from a point x to a given
% set of points S.
function dmin = distance_function (x,S)

dmin = inf;

for i=1:size(S,1)
   dx =  abs(S(i,1)-x(1));
   dy =  abs(S(i,2)-x(2));
   d  =  sqrt(dx^2+dy^2);
   if d<dmin dmin=d; end
end
```

Figs. (1) and (2) show the results of a couple of numerical examples. In both cases, the width and height of the domain are unity, and the mesh refinement, $h = 0.01$. The points in $\boldsymbol{S}$ are denoted by the green squares, and the contours give the distance function over $\Omega$.

If we are dealing with $M$ pieces of data in $\boldsymbol{S}$, and $N$ nodes in each spatial dimension, in 2D, we have a method that is $O(MN^2)$. This is clearly not optimal, so we turn to the next less expensive method, which is what is used in the current final implementation.

## 2.2 Viscosity solution of Eikonal equation

The following implementation is based primarily on the details presented in [Zhao, 2004]. A Gudinov upwind difference scheme is used to compute the viscosity solution $u(\boldsymbol{x}) \geq 0$ for the Eikonal equation,

$$\begin{aligned} |\boldsymbol{\nabla} u(\boldsymbol{x})| &= 1, \boldsymbol{x} \in \Omega \\ u(\boldsymbol{x}) &= 0, \boldsymbol{x} \in \Gamma \end{aligned} \tag{8}$$

As shown in class (on 10/11/2005), a field $u(\boldsymbol{x})$ that satisfies this differential equation and boundary condition is a measure of how far you are from a given curve $\Gamma$, and is the required distance function.

The algorithm is briefly described and implemented in 2D, but can be easily extended to higher dimensions. In the following, $h$ is the mesh size, $x_{i,j}$ is an arbitrary grid point and $u^h_{i,j}$ denotes the numerical solution at $x_{i,j}$. The *discretisation* used for Eq. (8) is the following upwind scheme,

$$[(u^h_{i,j} - u^h_{xmin})^+]^2 + [(u^h_{i,j} - u^h_{ymin})^+]^2 = h^2, \forall i = 2, \ldots, N-1; j = 2, \ldots, N-1, \tag{9}$$

where $u^h_{xmin} = min(u^h_{i-1,j}, u^h_{i+1,j})$ and $u^h_{ymin} = min(u^h_{i,j-1}, u^h_{i,j+1})$. One sided differences are used at the boundaries. The values of $u(\boldsymbol{x})$ are *initialised* to 0 for $\boldsymbol{x} \in \Gamma$, and large positive values elsewhere. (The current implementation does not handle interpolation of initial values close to grid points, but this can be added). Then, at each $x_{i,j}$ (not fixed initially), a solution $\bar{u}$ of Eq. (9) is found (using the unique solutions from [Zhao, 2004]). $u^h_{i,j}$ is updated to be the minimum of the previous $u^h_{i,j}$ and $\bar{u}$. Gauss-Seidel iterations with alternative sweep ordering are used to solve for the field everywhere in the domain. The particular sweep orderings used are indicated in the MATLAB code snippet below. An arbitrary final sweep is used to ensure convergence.

```
% Perform the Gauss-Seidel iterations

% First pass  -- Lower left to upper right
for i=1:nodey
    for j=1:nodex
        seidel
    end
end

% Second pass -- Lower right to upper left
for i=1:nodey
    for j=nodex:-1:1
        seidel
    end
end

% Third pass  -- Upper right to lower left
```

```
for i=nodey:-1:1
    for j=nodex:-1:1
        seidel
    end
end

% Fourth pass -- Upper left to lower right
for i=nodey:-1:1
    for j=1:nodex
        seidel
    end
end

% Fifth pass, to guarantee convergence.
% Arbitrarily chosen to be lower left to upper right
for i=1:nodey
    for j=1:nodex
        seidel
    end
end
```

The actual calculations carried out are detailed in the following subroutine.

```
% The Gauss-Seidel loop

% Only carry it out if the node hasn't been set initially
if ~(isinfound([i,j],found))
    if ((i>1)*(i<nodey))
        uymin = min(u(i-1,j),u(i+1,j));
    elseif (i==1)
        uymin = u(2,j);
    else
        uymin = u(nodey-1,j);
    end

    if ((j>1)*(j<nodex))
        uxmin = min(u(i,j-1),u(i,j+1));
    elseif (j==1)
        uxmin = u(i,2);
    else
        uxmin = u(i,nodex-1);
    end

    if (abs(uxmin-uymin)≥h)
        ubar = min(uxmin,uymin)+h;
    else
        ubar = (uxmin+uymin+sqrt(2*h^2-(uxmin-uymin)^2))/2;
    end

    u(i,j)=min(u(i,j),ubar);
end %if
```

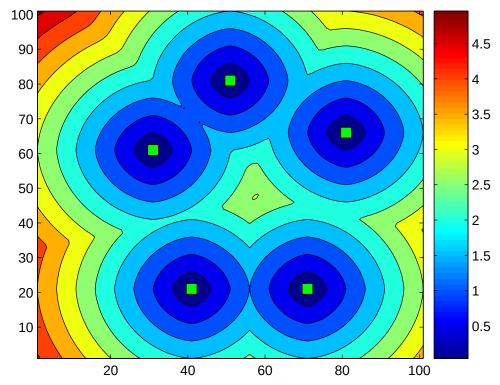

Figure 3: Distance function contours for an arbitrary set of 5 points.

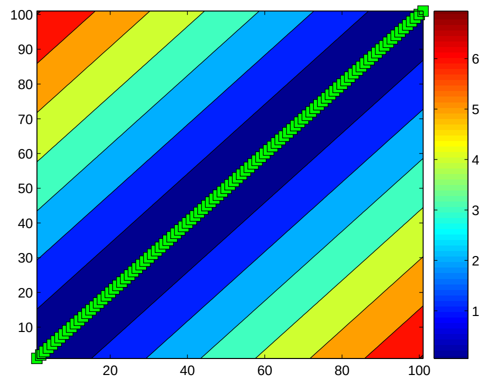

Figure 4: Distance function contours for $\boldsymbol{S}$ being the diagonal.

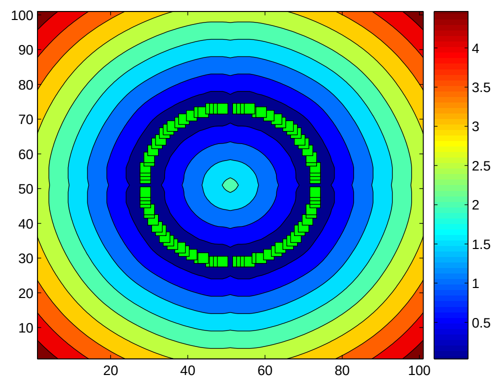

Figure 5: Distance function contours for a set of points forming a central circle.

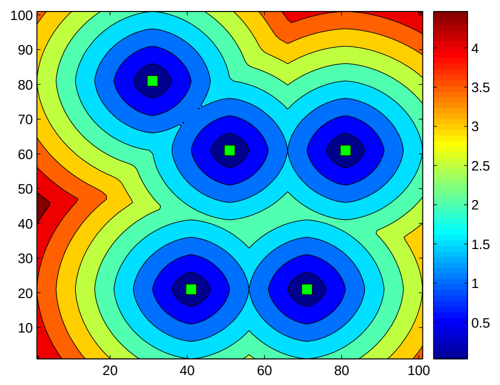

Figure 6: Distance function contours for an arbitrary set of 5 points forming a non-convex polygon.

In order to check the distance function evaluator, several numerical examples were run. Figs. (3–6) show the results for a few cases. In all cases, the width and height of the domain are 10, and the mesh refinement, $h = 0.1$. The points in $\boldsymbol{S}$ are denoted by the green squares, and the contours give the distance function over $\Omega$.

## 3 Evolving the level set

Now that we have determined the distance function $d(\boldsymbol{x})$ all over $\Omega$, we return to the level set equation, Eq. (6). The gradient terms, $\boldsymbol{\nabla}(\cdot)$, are computed using simple finite different schemes of varying orders of accuracy. Based on the size of the stencil used, the original data, $d^h_{i,j}$ or $\varphi^h_{i,j}$ (the numerical approximations to $d(\boldsymbol{x})$ and $\varphi(\boldsymbol{x})$ at $\boldsymbol{x}_{i,j}$) are suitably extended by extrapolation at the boundaries. The following MATLAB code snippet shows the simplest of these schemes.

```
% Determine the gradient using the stencil
%    |
% --u--
%    |

% u is the field extended by one node at the boundary

 for i=1:nodey
     for j=1:nodex
         gradu_y(i,j) = (u(i+1,j)-u(i-1,j))/(2*h);
         gradu_x(i,j) = (u(i,j+1)-u(i,j-1))/(2*h);
         norm_gradu(i,j) = sqrt (gradu_x(i,j)^2+gradu_y(i,j)^2);
     end
 end
```

Before we start the process, we need to determine a suitable initial value of the level set function, $\varphi(\boldsymbol{x})$. This implementation executes the simplest possible idea by first determining the bounding box of the points in $\boldsymbol{S}$, and making sure the zero level set of $\varphi(\boldsymbol{x})$ is that bounding box. The following is the relevant MATLAB code.

```
% Some random points
S=[4.,2.; 3.,6.; 5.,8.; 8.,6.5; 7.,2.;];

%Determine bounding box of point for a decent initial guess at the level
% set function
bbox = zeros(2,2);
bbox(1,:) = min(S)-h;
bbox(2,:) = max(S)+h;

phi = ones(nodey,nodex);
for i=1:nodey
     for j=1:nodex
         if (pos(i,j,1)≥bbox(1,1)&pos(i,j,1)≤bbox(2,1)...
                 & pos(i,j,2)≥bbox(1,2)&pos(i,j,2)≤bbox(2,2))
             phi(i,j)=-1;
         end

     end
end
```

Fig. (7) shows the original level set function $\varphi(\boldsymbol{x})$ for an example computation involving the five points denoted by green squares in Fig. (3). Fig. (8) shows the corresponding zero level set. The algorithm then reinitializes $\varphi$, retaining the bounding box as $\Gamma$, to a signed distance function and uses that as the initial value. This choice can be much more optimal, as in [Zhao et al., 2001].

Deriving from published implementations of the ideas in [Osher and Fedkiw, 2002], we proceed to solve Eq. (6) using standard schemes. With $d(\boldsymbol{x})$, and $\boldsymbol{\nabla}(d)$ found and $\varphi(\boldsymbol{x})$ initialised to a signed distance function resulting in the bounding box as its zero level set, the following algorithm is applied to evolve $\varphi(\boldsymbol{x})$ in time.

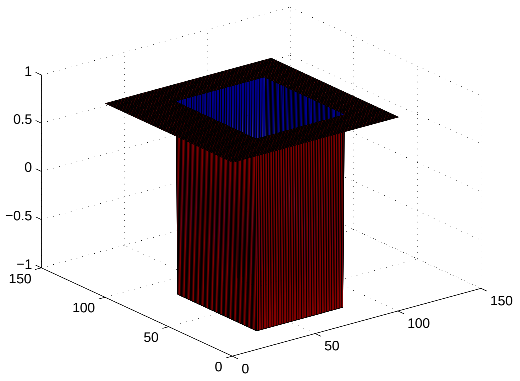
Figure 7: Original level set function, $\varphi(\boldsymbol{x})$.

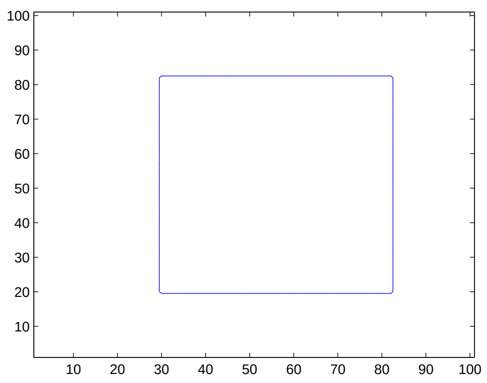
Figure 8: The original zero level set, $\Gamma$.

- Determine an appropriate Euler time-step size, $dt$ from the values of the vector field contributions $(\boldsymbol{\nabla}(d)_x, \boldsymbol{\nabla}(d)_y)$, curvature contributions $(d(\boldsymbol{x}))$ and mesh size, to ensure stability.
- Set $t = 0$
- Loop over the following for a set number of iterations
    - fgrad = Forcing contribution from the vector field, $\boldsymbol{\nabla}(d)$
    - fcurv = Forcing evolution from curvature-based force
    - Evolve the level set function as $\varphi = \varphi + (\text{fcurv} - \text{fgrad})dt$
    - $t = t + dt$

Additionally, we reinitialise $\varphi$ to a signed distance function at a certain frequency of iterations so that the Gudinov scheme used to implement the above pseudocode remains accurate. Figs. (9–12) show some representative numerical examples.

## 4 Potential improvements

Currently, the implementation starts the process with the most basic selection of the level set function $\varphi(\boldsymbol{x})$; one which defines the bounding box of the given set of points $\boldsymbol{S}$. However, the method can be made much more efficient if we start with a more precise guess, such as one obtained from the the tagging scheme presented in [Zhao et al., 2001], which is only $O(Nlog(N))$ in computational expense. Additionally, the current the process runs for a preset number of iterations based on crude numerical experimentation and visual inspection. This can be modified by calculating an error measure after each iteration, and stopping the process when it falls within the required tolerance.

When a point on $\boldsymbol{S}$ doesn't lie on a grid point, we need to interpolate the initial values of the distance function appropriately before solving the Eikonal equation, Eq. (8). This is

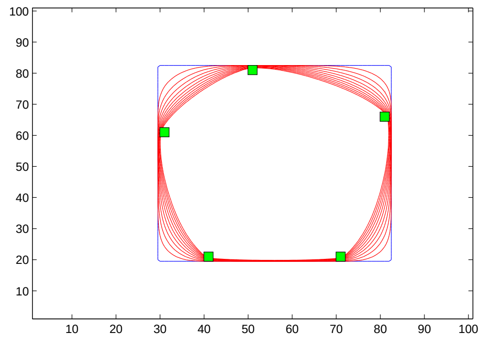

Figure 9: This figure shows snapshots of evolution of the zero level set at a certain frequency of iterations. The problem at hand is to determine a curve encompassing those 5 points marked by green squares. The results shown here are for a scheme reproducing what was done in [Zhao et al., 2001], which uses only the convection term. The curvature contribution is absent.

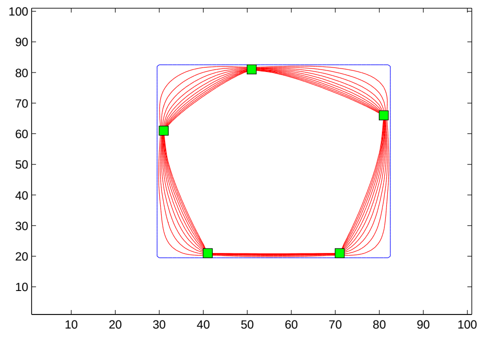

Figure 10: This figure shows the current algorithm incorporating the curvature term for the same problem. The snapshots are shown at the same frequency of iterations, and the zero level set seems to evolve slightly faster toward the points. In these plots, the initial curve is blue, and the red curves are subsequent iterations.

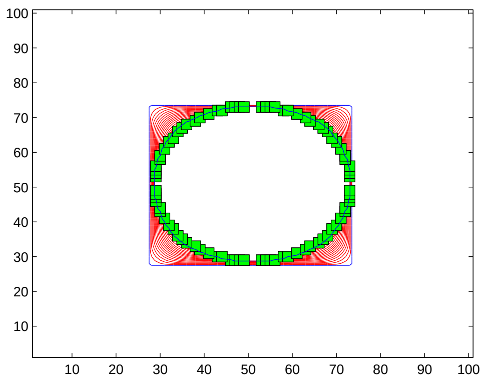

Figure 11: This is similar to Fig. (10), but there are many more points that the curve is aiming to wrap. The final zero level set curve $\Gamma$ is also denoted by a blue to delineate it from the numerous green squares.

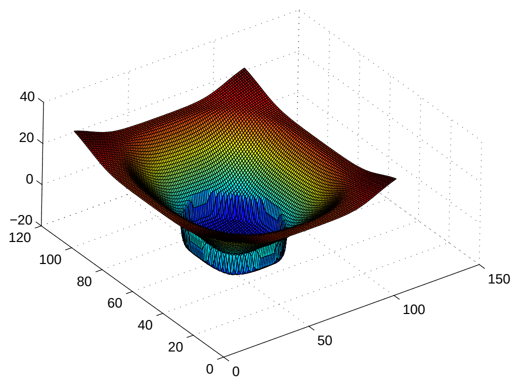

Figure 12: This figure shows the final value of $\varphi(\boldsymbol{x})$ corresponding to the last zero level set curve $\Gamma$ in Fig. (11).

currently not done, and the issue is avoided completely by suitably picking $\boldsymbol{S}$ such that all the points in it fall at grid points. This needs to be changed. With all these refinements worked in and functioning, it will be interesting to try other (potentially cheaper) schemes (such as the methods described in [Mauch, 2000, Wenwang, 2003]) for distance function evaluation and only local computation (within a narrow band of the zero level set) as described in [Peng et al., 1999], before moving the implementation over to 3D.

## References

[Mauch, 2000] Mauch, S. (2000). A fast algorithm for computing the closest point and distance transform. Technical report, Caltech ASCI technical report 2000.077.

[Osher and Fedkiw, 2002] Osher, S. and Fedkiw, R. (2002). *Level Set Methods and Dynamic Implicit Surfaces*. Springer.

[Peng et al., 1999] Peng, D., Merriman, B., Osher, S., Zhao, H., and Kang, M. (1999). A PDE-based fast local level set method. *Journal of Computational Physics*, 155(2):410 – 38.

[Russo and Smereka, 2000] Russo, G. and Smereka, P. (1 Sept. 2000). A remark on computing distance functions. *Journal of Computational Physics*, 163(1):51 – 67.

[Wenwang, 2003] Wenwang, Z. (2003). The fast sweeping method of Eikonal equations and its parallelism. Master's thesis, Royal Institute of Technology.

[Zhao, 2004] Zhao, H.-K. (2004). A fast sweeping method for Eikonal equations. *Mathematics of Computation*, 74(250):603 – 627.

[Zhao et al., 2001] Zhao, H.-K., Osher, S., and Fedkiw, R. (2001). Fast surface reconstruction using the level set method. *Proceedings IEEE Workshop on Variational and Level Set Methods in Computer Vision*, pages 194 – 201.