

Level Sets

Introduction

We're going to talk about level sets as a way of computing the evolution (or motion) of a curve over time. This is not because we're actually interested in curves moving over time, but because we are interested in finding curves that optimize some measure of goodness for boundaries. In this case, the evolution of the curve will just be a kind of gradient descent to an increasingly better boundary. However, I'll mention another big application of curve evolution in vision, which is to build a scale space for curves. In this case, the evolution of the curve can be a diffusion, that simplifies the curve.

Notation:

$C(s,t)$ is a point on the curve at some position and time. s denotes arclength, and t denotes time.

C_s is the tangent to the curve. The change in the position, as arclength changes.

C_{ss} is the curvature times the normal, the change in the tangent as arclength varies.

C_t is the motion of a point over time. Notice that C_t can have a component in the normal direction and the tangent direction, but we are only interested in the change in the normal direction. Changes in the tangent direction correspond to points moving along the curve, but to us all points on the curve will be the same.

Examples of simple curve evolution:

- 1) Evolution with the distance transform. Here a curve moves inward at a constant rate of speed. We can write this as $C_t = n$ (where n is the normal vector). This is an old curve evolution, introduced by Blum in the 60s and called the grassfire transform. This has been used for shape description, because the singularities of this transform intuitively describe the skeleton of a shape.

This simple transform illustrates a couple of difficulties raised by curve evolution. First, the topology of the curve can change. This is a big challenge for many numerical methods. If we think about the simple descent algorithm I described for SNAKES, one can see that it is totally unsuited to handling changes in topology. Another way to think about this is if we evolve the curve according to $C_t = n$, then the curve self-intersects after a while.

A second problem is that we can start with a simple smooth curve, and a simple evolution, and wind up with discontinuities. In this case, this can happen when a point is equidistant to two (or more) points on the initial curve, and these are the closest points on that curve.

- 2) In a second flow, the motion is proportional to curvature: $C_t = kn$. This flow has some very interesting properties. First, we can rewrite this as: $C_t = C_{ss}$. This should remind you of the diffusion equation. This has the property that it steadily reduces the curvature of the curve until it is convex, then ultimately smooths it to a point, without self-intersections ever occurring. This is like what happens with diffusion of a function, but with a curve embedded in 2D.
- 3) Just as a minor note, these flows can be used to smooth images. We take the level sets of the images, treat these as contours, and then evolve these with a curve smoothing method.

Level Set Formulation

Now we formulate curve evolution.

Let F be some function that describes the speed of the curve flow in the direction normal to the curve. F may be a function of x , y and may depend on image properties. It may also depend on things like the curvature of the curve.

$T(x,y)$ is the time when the curve crosses the position (x,y) (note that this assumes the curve crosses each point once).

Then we have:

$|\text{grad } T| F = 1$ with $T(x,y) = 0$ on the initial curve. Boundary Value Formulation

This just says that the faster the curve is moving, the slower the arrival time is changing.

One way to write this if the arrival time isn't a function of x and y , is to formulate a function ϕ of x , y , and time, so that the 0 set of this function represents the position of the curve. This is just like writing an implicit equation for a circle, or any other curve.

Let $x(t)$ be a particle on the curve, whose position is a function of t . Then we must have:

$$\phi(x(t), t) = 0.$$

By the chain rule we have:

$$\phi_t + \text{grad } \phi(x(t), t) \cdot x'(t) = 0$$

Since F is the speed in the normal direction, we have:

$$x'(t) \cdot n = F, \text{ where } n \text{ is } \text{grad}(\phi)/|\text{grad}(\phi)|$$

Substituting, $x'(t) \cdot \text{grad}(\phi) = F|\text{grad}(\phi)|$ we get:

$\phi_t + F|\text{grad } \phi| = 0$, with the initial condition $\phi(x,t=0)$. INITIAL VALUE FORMULATION

I'm skipping over a couple of points here. We haven't really said how to get the full initial condition. And, if we want F to be a function of curvature, we have to write curvature as a function of ϕ for this formulation. That is, we need an expression that evaluates to curvature on the zero level set.

Both of these formulations can be used to solve the problems of curve evolution. Each of these give us PDEs that we want to solve numerically. In both cases we need to worry about singularities. In both cases, efficiency comes from noticing that we care about information initially along a curve, and we need to propagate this information in the neighborhood of the curve.

Discontinuity of Solution

As we saw last time, for some simple curve evolutions, we can obtain curves with discontinuities. When we formulate curve evolution with a PDE, this PDE will not be defined at discontinuities, since we can't take derivatives. However, when we evolve a curve with a speed proportional to the curvature, we don't get this problem, the curve only gets smoother. It turns out that if we take other curve evolutions, and add a small component of this diffusion, (this diffusion force times epsilon) our curve will stay smooth. This makes sense, since as the curve gets singular, the curvature goes to infinity, so even when epsilon is small, this smoothing has a big effect. Then, we take the limit of this process, as epsilon goes to 0. This gives us well-defined behavior, and converges to what we would expect. There are technical conditions that ensure that the algorithms that solve level sets have this behavior, but I'll ignore those.

Solution Methods

The solution methods we will describe depend on the notion of upwind differencing, so let me explain that in a simple setting. The intuition is that curve evolution is an *advection* equation, in which something is moving. In that case, it is important to base your numerical approximations to derivatives on the region where the thing is coming from, not on where it's going to.

Consider $u_t(x,t) + u_x(x,t) = 0$, with $u(x,0) = f(x)$.

The solution to this equation is constant along parallel lines in x - t space.

We know that we can approximate these derivatives numerically as:

$$u_t = (u(x, t+k) - u(x,t)) / k$$

$$u_x = (u(x+h,t) - u(x,t))/h$$

Solving the equation with this, we get:

$$u(x, t+k) = -k(u(x+h, t) - u(x, t))/h + u(x, t).$$

The problem with this is that we base the value of $u(x, t+k)$ on values $u(x, t)$ and values to the right of that, when the real solution depends on values to the left. So in upwind differencing, we just instead use:

$$u_x = (u(x, t) - u(x-h, t))/h.$$

This is easy, because it's clear the direction in which information is propagating in this simple example. With curve evolution, the direction of propagation depends on where we are in the curve, and exactly how things are propagating. So we need an upwind differencing scheme in which the direction of propagation is adaptive. This leads to the fast marching method, which we'll look at for the first curve evolution we discussed, $|\text{grad } T| F = 1$. In this case, we need to compute, for every discrete location, the time when the curve reaches that location. We start with a set of nodes that have a time of zero. At every time step, we update those nodes that can be computed with an upwind scheme. Then we keep the one for which the time is smallest, and try again on all nodes.

This should look a lot like Dijkstra's shortest path algorithm. And, in fact, this problem is really just a shortest path problem. The difference is that we update times using the underlying PDE. This allows us to interpolate times, and avoid some discretization effects. For example, in the limit as the grid gets small, this method converges to an exact solution, whereas Dijkstra wouldn't.

We can also solve the level set formulation more efficiently, using a similar idea. Here, we are computing ϕ as a function of x and t . But we are only interested in the solution near the zero level set. So, at $t = 0$, we form a narrow band around the level set, and let the solution propagate upwind, until we leave this band. Then we find the zero level set, and reformulate a new problem.