Multi-Region Active Contours with a Single Level-Set Function

Submitted by : Chen Shapira and Tamir Segev
Advisors : Anastasia Dubrovina and Yaron Honen

March 2016
# Table of Contents

Abstract .............................................................................................................................. 3  
Introduction & Theoretical Background ........................................................................... 4  
Proposed Approach .......................................................................................................... 7  
Implementation .................................................................................................................. 9  
Results ............................................................................................................................... 11  
Future Work ....................................................................................................................... 17  
Summary ............................................................................................................................ 18  
References ......................................................................................................................... 19
Abstract

Image segmentation is considered an important problem in image processing. Over the years, many approaches have been proposed to solve this problem. While some of these approaches provide exquisite theoretical solutions, they are usually not practical.

In our work, we’ll focus on the Multi-region Active Contours with a single Level-Set function method proposed in [1]. This method allows quick & accurate image segmentation on 2d and 3d images. This is done by dividing the image into multiple regions and iteratively calculating a single non-negative distance function, which is easily extracted using the Voronoi Implicit Interface Method.

Here, we implemented the proposed method in C++ using the commonly used library - OpenCV, allowing this implementation to be easily ported across multiple platforms, and easily used. This will provide users the ability to apply this segmentation method in a relatively fast and agile environment.
Introduction

2.1 Image Segmentation

Image segmentation is a process in which one partitions an image into several regions. Segmenting an image is commonly used to define objects and detect their boundaries in an image. In most cases we’re interested in segmenting the image into disjoint regions, that put together, comprise the entire image. Image segmentation is a major part of the field image processing. It’s a tool used in objection detection, classification, computer vision, machine learning and many other areas. Several approaches exist for image segmentation, and can be classified by the following traits: automatic vs. assisted, object & background vs. multi-region, different criteria, prior data and training sets, and many other criteria.

2.2 Active Contours

In our project, we used the approach of Multi-Region Active Contours with a Single Level Set Function, as proposed in [1]. In the active contours method, we model the region boundaries as curves (contours), while choosing a segmentation criteria to minimize, using an energy functional. This allows minimization of arbitrary energy functionals while maintaining sub-pixel precision of boundaries.

2.3 Energy Functional

The method we use require us to minimize the following energy functional:

\[ E(C) = E_{data}(C) + \mu E_{reg}(C) \]
where: $C(s) = (x_1(s), x_2(s))$

We notice the energy functional is dependent on the contour $C$. We want to find $C$ that will minimize this energy functional.

The term $E_{\text{data}}(C)$ describes the part of the energy functional that is based on the image pixel values, such as described in [2],[3].

The term $E_{\text{reg}}(C)$ describes contour properties, such as contour length, and can also contain information regarding the image itself, as in the geodesic active contours model in [4].

We minimize the energy functional by using the steepest-descent method, with respect to the virtual parameter ‘$t’$, to receive:

$$C_t = -\delta E(C) / \delta C = -\left[ \frac{\delta E_{\text{data}}(C)}{\delta C} + \mu \frac{\delta E_{\text{reg}}(C)}{\delta C} \right] = F n$$

### 2.4 Geodesic Active Contours

The regularization term used for Multi-Region Active Contours is the Geodesic Active Contour regularization term:

$$\frac{\mu}{2} \sum_i \int_{C_i} g(C_i(s)) ds$$

This term is defined by $g(x)$, as suggested in [4]:

$$g(x) = (1 + |\nabla (G*I)|^2)^{-1}$$

This method’s regularization term allows minimization of the curvature of the contour.

### 2.5 Level-set Framework

Level set method is a framework for using level sets as a tool to analyze surfaces and shapes. In two dimensions, the level set method amounts to representing a closed curve $C$ using an auxiliary function $\phi$, called the level set function, as the zero level set of $\phi$:

$$C = \{(x,y) | \phi(x,y) = 0\}$$
The level set method manipulates $C$ implicitly through the function $\phi$. Level sets are usually used toward two-region image segmentation, though various attempts have been made to overcome this - most of them using multiple level set functions, such as [5]. In our work, as in [1], we use a single level set for multi-region segmentation, using a single non-negative level set function. This allows segmentation of an arbitrary number of regions. This is done by using the Voronoi Implicit Interface Method [6] [7].

### 2.6 Voronoi Implicit Interface Method

The Voronoi Implicit Interface Method is a framework for numerical solution of interface propagation problems, allowing numerous regions, arising from the field of computational fluid dynamics. This method allows the evolution of the level set function, by:

1. Finding the epsilon-level sets of the level set function $\phi$
2. Computing the Voronoi regions of each epsilon-level set
3. Reconstructing the contour $C$ as the intersection of the Voronoi regions

Using the tools we described above, we now suggest the following approach to solving the image segmentation problem - the Multi-Region Active Contours using a Single Level Set Function method.
Proposed Approach

3.1 Piecewise-Constant Model
In our project, we decided to use the piecewise constant model from [3] - assuming gaussian probability distributions, given by \( I \sim N(c_i, \sigma^2) \). From this we derive:

\[
E_{data}(C, \{c_i\}) = \sum_{i=1}^{M} \int_{\Omega_i} (I(x) - c_i)^2 \, dx
\]

In our implementation we decided to limit our problem to \( M=2 \) (which is also known as the Chan-Vese Model [3]). This means that the energy of a pixel is derived from the region the pixel is in, and the closest neighboring region to it.

Deriving the energy functional as seen earlier leads to the new contour evolution rule:

\[
\phi_t = \sum_{i \in N(x)} \left[ \left( I(x) - c_i \right)^2 + \frac{\mu}{2} \left( \kappa_i \cdot \langle \nabla g, n_i \rangle \right) \right] n_i = \sum_{i \in N(x)} \left[ F_{i_{data}}(x) + \frac{\mu}{2} F_{i_{gac}}(x) \right] n_i
\]

where \( c_i = \int_{\Omega_i} I(x) dx \int_{\Omega_i} dx \).

3.2 Extended Velocity Calculation
As seen in [8], evolving the contour \( C \) is equivalent to evolving the level set function:

\[
\phi_t = F_{ext} \left| \nabla \phi \right|
\]

where \( F_{ext} \) is the smooth extension of the force \( F \) to the entire domain, or at least a narrow band around \( C \). This emphasizes the “implicitness” of our method - we derive \( C \) implicitly from the evolution of \( \phi \).

Hence, in order to use the level set method, we must extend the contour velocity to the entire domain.

The extension velocity of the data term is:

\[
F_{ext}^{data}(x) = F_i(x) - F_j(x) = \left( (I(x) - c_i)^2 - (I(x) - c_j)^2 \right)
\]

where \( i \) is the region containing ‘x’, and \( j \) is the nearest neighboring region of ‘x’.

The extension velocity of the regularization term is, as derived in [4]:

\[
F_{ext}^{gac}(x) = div \left( g(x) \frac{\nabla \phi}{|\nabla \phi|} \right)
\]

Combining these two, we get:

\[
\phi_t(x) = \left[ F_i(x) - F_j(x) + \mu div \left( g(x) \frac{\nabla \phi}{|\nabla \phi|} \right) \right] |\nabla \phi|
\]

In order to evolve \( \phi \) from \( \phi_t \), we use semi-implicit LOD scheme as described in [9].

3.3 Fast Marching Method
This method is a numerical method introduced in [10], for solving boundary value problems. The algorithm is:

1. Assign to all pixels adjacent to the contour the color ‘black’, and calculate their distance to the contour (in a sub-pixel resolution).
2. Assign to all neighbors (4-way) of the black pixels the color ‘red’, and insert an evaluated
distance into a minimum heap.
3. Assign the color green to the rest of the pixels in the image, with their distance set to infinity.
4. Perform the following evolution until there are no more green pixels left:
   1. Choose the smallest pixel in the heap. If its color is black, continue to next iteration.
   2. Otherwise (pixel is red), assign its distance as the evaluated distance and change color
to black.

3.4 The Algorithm
Starting with an initial contour $C = C_0$, we compute an unsigned distance level set function
$\phi(x) = \phi_0(x)$ from the initial contour.
Then, perform the following sequence until convergence:
1. Calculate the extension velocity $F_{\text{ext}}$ according to the equation above, and calculate $\phi_t$.
2. Evolve $\phi$ from $\phi_t$ using the LOD scheme
3. Extract the epsilon level-sets of the evolved $\phi$.
4. Calculate the Voronoi Regions of the epsilon level-sets.
5. Reconstruct the evolved contour $C$ as the boundaries between the regions.
6. Perform re-distancing and calculate the iterations $\phi$ using the contour $C$ and the Fast
Marching Method on it.

Note that the algorithm described can also be run only on a narrow-band around $C$ instead of
the whole domain.
Implementation

4.1 C++
The algorithm proposed in the sections before, is limited by its runtime, which can prove to be a major obstacle. Due to this, we decided on implementing our project in C++. This allowed us to achieve better performance compared to other possible languages, such as Matlab, and gave us the ability to control our run-time memory allocation. The latter can be a major consideration, if our implementation is expanded to 3D images, as we shall discuss in "Future Work".

4.2 OpenCV
To assist us in implementing the various procedures in our project, we decided on using the open-source library OpenCV 3.0. OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at real-time computer vision, originally developed by Intel. We used this library extensively in our project, especially for defining and allocating our main structures - the matrices (images). Every operation done on matrices is already implemented, tested, and optimized in this library. For instance, matrix multiplication (and many other operations) are implemented in a multi-threaded manner, significantly improving the run-time of such an operation.
We also used OpenCV's built-in image processing methods, such as:
- Sobel - we used the Sobel Filter to calculate the image's gradient.
- distanceTransform - this method allowed us to easily calculate the distances of the pixels to the closest border pixels.
- GaussianBlur - This method assisted us when trying to calculate the regularization term.

4.3 The algorithm’s implementation
We implemented the algorithm discussed in 3.4 in the following manner:
Step 0: The Initial Contour mentioned in this step is implemented in the module “initial_contour_masks”. By default, the contour is a set of circles spanning the image. From this contour, we calculate the initial level-set function. This is done in the module “distance_funcs”.
Step 1: The extended velocity function is calculated in the module “force_funacs” and in “weight_funacs”, implementing the data and regularization terms, respectively. By default, the force function is derived from the piecewise constant model - meaning, it implements the Chan-Vese model with GAC as the regularization term, as discussed earlier.
Step 2: The evolution of phi is done in the module “evolve_funacs”. This is done using the semi-implicit LOD scheme, calculated on a three-diagonal matrix defining the derivation operator.
Steps 3-6: Epsilon level set extraction, Voronoi regions calculation, contour reconstruction, and redistancing of phi are all calculated in the module “redistancing_funacs”. This module also contains the implementation for the "Fast Marching Method", which assists in the final calculation of phi in each iteration.
4.4 Generalization of the algorithm
In addition to the default implementation, we added extensive support for easy generalization of the methods we chose. One possible extension we allowed is the addition of a different energy functional - our implementation is generic, in the sense that the force object used to calculate the extended force, is an abstract class with a pure-virtual method in (all in C++), and the “main” function passes an instance of the “Chan-Vese method” object, which implements its force function. This allows a user to implement his own extended force function, deriving from the energy functional of his choice.
In addition, we also implemented a special interface with the commonly used MATLAB tool. This allows users to implement in MATLAB their energy functional (more specifically, the force function derived from it), and use it seamlessly in a script we wrote. We also provided an example of this.
To allow even more customization on the user end, we added a special interface, where the user can draw his own initial contour instead of the default (set of circles). As we show in the results section, even a very inaccurate sketch of this contour improves results drastically.
Results

5.1 Segmentation Examples

When running our code on the following image ('obama.jpg' from the project examples directory), we obtain the following segmentation results:

![Image of Obama](obama.jpg)

<table>
<thead>
<tr>
<th>Original</th>
<th>1st iteration</th>
<th>2nd iteration</th>
<th>5th iteration</th>
<th>10th iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="obama_1st.jpg" alt="Image 1" /></td>
<td><img src="obama_2nd.jpg" alt="Image 2" /></td>
<td><img src="obama_5th.jpg" alt="Image 3" /></td>
<td><img src="obama_10th.jpg" alt="Image 4" /></td>
<td></td>
</tr>
</tbody>
</table>

The level-set function for each iteration:

<table>
<thead>
<tr>
<th>1st iteration</th>
<th>2nd iteration</th>
<th>5th iteration</th>
<th>10th iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="obama_ls_1st.jpg" alt="Image 5" /></td>
<td><img src="obama_ls_2nd.jpg" alt="Image 6" /></td>
<td><img src="obama_ls_5th.jpg" alt="Image 7" /></td>
<td><img src="obama_ls_10th.jpg" alt="Image 8" /></td>
</tr>
</tbody>
</table>

Note that in the level-set results, the gray-level of a pixel is its level set - the contours in each iteration are the “black” pixels.

Running on the following image ('flower.png'), we received the following results:

![Image of Flower](flower.png)

<table>
<thead>
<tr>
<th>Original</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="flower.jpg" alt="Image 9" /></td>
</tr>
</tbody>
</table>
Running on the following image ('sketch_color.png'), we received the following results:

These result align with our expectations - a good segmentation for these images is found using our algorithm. As expected from a steepest-descent optimization method, we can see that most of the advancement of the segmentation is achieved at the first iterations, and the more we get closer to the optimal solution - the less the algorithm advances.
5.2 Segmentation Performance
Our implementation can segment images of different sizes. It is expected that for larger images, the run-time will be (almost) linearly longer, with respect to the number of pixels in the image. This can be seen when viewing the segmentation of the following image (‘clinton.jpg’), with different sizes:

![Image of Hillary Clinton with segmentation results](image)

When segmenting this image over different image sizes, we received the following results (in seconds):

<table>
<thead>
<tr>
<th>Iteration \ Image Size</th>
<th>200x200</th>
<th>400x400</th>
<th>720x720</th>
<th>1200x1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.55385</td>
<td>8.84214</td>
<td>19.4748</td>
<td>40.2629</td>
</tr>
<tr>
<td>2</td>
<td>1.85931</td>
<td>5.66848</td>
<td>17.1476</td>
<td>43.5463</td>
</tr>
<tr>
<td>3</td>
<td>1.62422</td>
<td>5.4637</td>
<td>16.3112</td>
<td>34.4325</td>
</tr>
<tr>
<td>4</td>
<td>1.3332</td>
<td>5.3024</td>
<td>15.0653</td>
<td>33.546</td>
</tr>
<tr>
<td>5</td>
<td>1.34812</td>
<td>4.84125</td>
<td>13.9251</td>
<td>33.37</td>
</tr>
<tr>
<td>6</td>
<td>1.29236</td>
<td>4.4671</td>
<td>14.3317</td>
<td>33.8212</td>
</tr>
<tr>
<td>7</td>
<td>1.28993</td>
<td>4.03684</td>
<td>13.7124</td>
<td>31.649</td>
</tr>
<tr>
<td>8</td>
<td>1.35421</td>
<td>4.24658</td>
<td>12.8794</td>
<td>31.8627</td>
</tr>
<tr>
<td>9</td>
<td>1.33509</td>
<td>4.4917</td>
<td>13.5365</td>
<td>31.6875</td>
</tr>
<tr>
<td>10</td>
<td>1.28264</td>
<td>4.49044</td>
<td>12.1101</td>
<td>33.1953</td>
</tr>
<tr>
<td>11</td>
<td>1.3427</td>
<td>4.39936</td>
<td>12.5372</td>
<td>35.0835</td>
</tr>
<tr>
<td>12</td>
<td>1.32035</td>
<td>4.3532</td>
<td>12.7796</td>
<td>32.6506</td>
</tr>
<tr>
<td>13</td>
<td>1.25789</td>
<td>4.84434</td>
<td>12.1706</td>
<td>35.5387</td>
</tr>
<tr>
<td>14</td>
<td>1.07976</td>
<td>4.11092</td>
<td>11.4616</td>
<td>35.3814</td>
</tr>
<tr>
<td>15</td>
<td>1.07421</td>
<td>4.37707</td>
<td>13.2785</td>
<td>32.7203</td>
</tr>
<tr>
<td><strong>Elapsed</strong></td>
<td><strong>21.5636</strong></td>
<td><strong>74.311</strong></td>
<td><strong>211.56</strong></td>
<td><strong>520.628</strong></td>
</tr>
</tbody>
</table>
These results show that, as the same image is taken in several different sizes, the total time of the segmentation changes linearly with the size of the image. This can be expected, since the algorithm runs (almost) linearly with respect to number of pixels in the image - $O(n\log n)$.

Another important result that is presented in the table, is the improvement of the run-time of each iteration, when comparing the first iterations to later iterations. For example, the first iteration of the 400x400 image takes 8.8 seconds, compared to the 10th iteration, which takes 4.49 seconds - almost half the time. This also can be expected, since steepest-descent approaches converge slower when approaching the minima, thus, the descent is milder in each iteration, and can be calculated faster. This can also be attributed to the reduction in the number of regions when approaching convergence in most cases.

While trying to improve performance, we profiled our system using “Instrument’s Time Profiler” for OS-X. This allowed us to find bottlenecks in our implementation. We noticed that largest chunk of time was consumed by the “perform_redistancing” module - the module responsible for calculating stages 3-6 in our algorithm. For example, these are the profiling results for ‘sketch_color.png’:

As you can see, this module takes 91.4% of run-time. When analyzing this in a finer granularity, we concluded that this is due in large to the “Fast Marching Method” as we implemented it. Several attempts were made to improve this result:
- Using a Fibonacci Heap instead of a Minimum Heap through the Boost Library
- Using a Linked List instead of a Heap
- Pre-allocating a buffer (an array of data structures) for instances which otherwise would be allocated every iteration of the Fast Marching Method.

Unfortunately, none of these attempts showed any significant improvement in the run-time of the algorithm.

5.3 Limitations
Our algorithm, as strong as it is, is limited in several aspects, and does not always converge well. As with other examples of gradient-descent algorithms, our algorithm can also converge to
a local minima instead of the global minima. An example can be seen here - the algorithm converges to a local minima after two iterations, with a poor result:

These results can be explained by the very mild gradient, caused by the halo of the moon. The gradient is an important component in the evolution of the contour in our method.

Another limitation is the choice of an initial contour. As explained earlier, the choice of a good initial contour can be crucial to producing good results. The default initial contour is a set of circles, which is not ideal for all images (though, as we saw earlier, for most cases is good enough). An example can be seen here - the initial contour produces a result which does not have contours at all, which is presented as a single label segmentation:

In contrast to this, when we choose a custom initial contour using the extension provided with our implementation, we achieve much better results:
These limitations and others, can be overcome using the extensions we provide. Using our project, one can define and create an Energy Functional which overcomes these limitations. An example of such a functional is the pairwise constant model, described in detail in [1].
The algorithm we implemented in our project is just a first taste of what can be done using the “Multi-Region Active Contours with a Single Level Set Function” method. A trivial expansion that is introduced and shown in [1] is the expansion from 2D to 3D images. This allows the production of image segmentation using volumetric data, which has important significance in the medical world.

Our implementation consists of a single energy functional - the piecewise constant model - which is based heavily on the difference in color between different segments. This can be very limiting, especially in images with a low variance in color. Other functionals include, but not limited, to those proposed in [1], such as, the pairwise constant model. These can easily be implemented using the expansion we provide.

As shown earlier, the choice of an initial contour can be of great significance. Improper initial contours can lead to convergence to local minima, or even no convergence at all. As convenient as it may seem, the ability to define a custom contour is not always practical. Thus, different criteria and methods for the choice of an initial contour can be suggested - such as, segmenting initially by the minimizing the M.S.E of a quantization of the image using N colors, and segmenting initially according to similar colors (known as the K-Means Algorithm).

In the results section, we show that the main bottleneck of our algorithm lies in the “Fast Marching Method”, which is responsible for over 90% of the run-time. On large 2D images, this can prove to be infeasible and unusable in any practical manner. Using a different method to calculate the distances, perhaps in the expense of a less accurate method, can be used instead.

Another interesting possibility is converting our implementation to the mobile market, which is the largest market today. Specifically, all the components of our implementation can be easily converted to be used on an Android phone. This will allow millions of users to automatically segment images using our project every day.
Summary

Overall, we’ve seen good results for segmenting many types of images. In many cases we tested, we saw that the segmentation and level-set function that evolved during the iterations were as expected, and very close to the contours perceived by the naked eye.

We saw the limitations of our algorithm when we tried segmenting images with a mild gradient (such as the aura of the moon). We also noticed that we can improve the segmentation of some images by choosing a custom initial contour for them, and offered a few ideas on how to expand our implementation.

In a personal note, we would like to add that the project introduced us to the wonderful world of image processing and computer vision. We’ve gained much knowledge of this field, thanks to this project, and contributed to our understanding of image segmentation and image processing in general.
References


