Implementation of an Image Inpainting Algorithm Based on the Fast Marching Method

Ade Thornhill, Jonas Vinson, Miles Weberman

December 2022

Abstract

In this paper we provide an implementation of an image inpainting algorithm based on Alexandru Telea's 2006 publication titled "An Image Inpainting Technique Based on the Fast Marching Method" [1]. We will give an overview of different approaches to image inpainting and the advantages and disadvantages of using the Fast Marching Method (FMM). We will discuss our (Python) algorithm implementation and compare it with existing implementations (such as the one in OpenCV). We will provide numerical and visual comparisons between both algorithms' performances.

1 Introduction

Image inpainting is a technique where missing or damaged parts of an image are filled in using the surrounding data such that the completed image looks realistic. Image inpainting originated in the 1800s with the restoration of physical artwork. The technique has evolved to be an important tool in the fields of digital art and image processing, and continues to be an active area of research and development in Computer Vision.

Many methods have been proposed and can (for the most part) be categorised into 3 techniques: sequential-based, CNN-based and GAN-based. Sequential-based methods can further be categorised into patch-based and diffusion-based methods [2]. Patch based methods fill in missing regions by finding well matching candidate matches. Diffusion based methods propagate the boundary of missing regions inwards until all pixels are inpainted.

Since Telea's article was published, many other inpainting algorithms have been proposed. The main difference between Telea's algorithm and the newer diffusion based methods is that the new implementations have much more complex constraints on the colour, texture, brightness, etc. of the pixel to be inpainted. CNN methods are based on ML algorithms. They aim to train a regressor with the masked image as input to generate the original image. After training with many images and many masks the model will learn to produce the undamaged image from the damaged one.

In this paper, we will be diving into a diffusion-based method, particularly an algorithm proposed by Alexandru Telea in his paper "An Image Inpainting Technique Based on the Fast Marching Method". The main idea behind this method is to inpaint pixels at the boundary between the known and unknown regions of the image and propagate the boundary of the unknown region inward. The order in which the boundary pixels are inpainted is determined by solving the Eikonal equation at those boundary points. This is done efficiently by implementing the FMM. The Eikonal equation is a partial differential equation which behaves similarly to the propagation of wave fronts, and is given by

$$|\nabla T| = 1 \tag{1}$$

1.1 Previous Related Work

The idea of using PDEs which arise in fluid dynamics to determine the order of diffusion of the unknown region, originated in a publication by Bertalmio et al [3]. The authors present a method in which isophote lines are continuously propagated from the exterior into the unknown region of the image. Isophotes are lines on a surface which connect points (in this case pixels) of constant intensity. These are also called level lines. In Bertalmio's paper, Navier-Stokes equations are utilised to ensure a smooth propagation of the isophotes. Although this yields realistic results, solving Navier-Stokes is a complex problem leading to difficult and computationally expensive implementations.

In this algorithm, image smoothness information, which is estimated by the image Laplacian, is propagated along lines of similar direction to the isophotes (lines of constant brightness). The isophotes are estimated using the image gradient information.

Other similar methods which involve PDEs have been proposed such as the Total Variational (TV) model [4] (which utilises a Euler-Lagrange equation). Or the Curvature-Driven Diffusion (CCD) model [5], which improves upon the TV model.

The above methods have several drawbacks that hinder their use in practice. The PDE-based methods require implementing complex iterative numerical methods and techniques, such as anisotropic diffusion and multiresolution schemes [1]. Additionally, such methods are quite slow. The above shortcomings led to the proposal of Telea's method which was a new inpainting algorithm based on propagating an image smoothness estimator along the image gradient.

This algorithm provides the following advantages over above methods:

- It is simple to implement [1], which can be considered this algorithm's most attractive feature. We prioritised preserving this in our implementation.
- It is considerably faster than other inpainting methods [1]. This is due to the fact that the FMM is an efficient way of solving Eikonal equations.
- It produces similar results compared to the other methods [1].

2 Our Method

The implementation of the method was done in Python. The popularity of Python in computer vision and the absence of a good open-source Python implementation are the main reasons that led us to this decision. The link to the implementation is provided at the end of the paper.

Our inpainting method takes as input an image with missing information, and a mask which specifies the region that needs to be inpainted. This is standard for most inpainting methods. We also allow the user to select the size of the neighbourhood used when inpainting a pixel.

The implementation is based largely on the description and pseudo code from Telea's original paper [1]. It can be resumed in 2 phases: initialization and propagation. First, each image pixel is assigned a flag describing the pixel's position relative to the inpainting region. This flag can assume three values:

- A pixel in the known region of the image is assigned KNOWN
- A pixel in the unknown region (i.e. the part that needs to be inpainted) is assigned INSIDE.
- Pixels on the boundary between the known and unknown parts of the image are set to BAND.

We initialize the distance map T which represents the solution to the Eikonal equation (Equation 1). We set each KNOWN and BAND pixel to zero. Each INSIDE pixel is set to 10^6. Finally, BAND pixels are added to a heap with T as the node key. For this, we used the built-in Python package heapq which is implemented as a priority queue (pushing and popping run in $O(\log n)$). The FMM method will inpaint points in order of their distance T which is why using a heap is efficient.

One design choice we made was that BAND pixels are already inpainted. It is also possible to implement the algorithm such that points on the boundary are not inpainted, this leads to similar results [1]. However, we obtained slightly better results with this implementation. This means that when a BAND pixel is popped from the heap, its neighbours in the unknown region (INSIDE pixels) are inpainted, set to BAND, and subsequently added to the heap.

The propagation phase inpaints pixels iteratively in increasing order of their distance to the boundary given by T, which is the solution of the Eikonal equation. The procedure can be resumed in 4 steps which are repeated until all points are inpainted:

1. Pop from the heap the boundary pixel p with the lowest T value and update its flag to KNOWN.
2. Inpaint the neighbours of p that are labelled INSIDE as described in section 2.1.
3. For neighbours of p that are flagged as INSIDE or BAND, update their value T by solving the Eikonal equation as detailed in section 2.2.

4. Finally, insert the newly inpainted points to the heap and change their flag to BAND. The points that were already flagged BAND are reinserted into the heap with updated T values.

2.1 Inpainting a pixel

Inpainting a pixel p is done as a function of a Neighbourhood B_ϵ around p (appropriate values of ϵ are discussed in section 3). In the original implementation of the algorithm, the neighbourhood consists of points at a distance, ϵ or less, that are on the same side of the boundary as the pixel p, that is, the neighbourhood is shaped like a half-circle. Our implementation considers points that are on each side of the boundary as long as they are within distance, ϵ or less from p. As the inpainting region becomes narrower, the neighbourhood B_ϵ is shaped like a circle centered at p with radius ϵ (see Figure 1). This ensures the resulting image has fewer discontinuities in the inpainted region.

Once the neighbourhood B_ϵ of a point p has been calculated, the image pixel

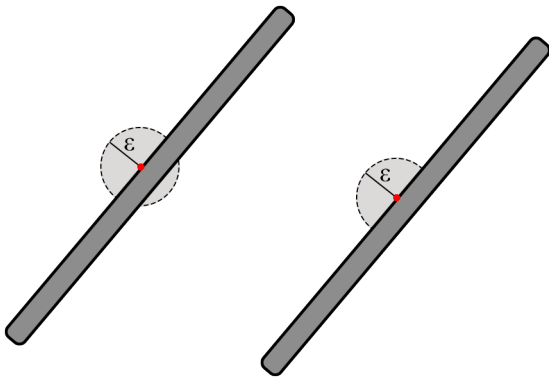

Figure 1. The neighbourhood B_ϵ around the point to be inpainted in our implementation vs the original implementation

is inpainted for each RGB colour channel according to the following equation:

$$I(p) = \frac{\sum_{q \in B_\epsilon(p)} |w(p,q)| I(q)}{\sum_{q \in B_\epsilon(p)} |w(p,q)|} \tag{2}$$

The weighting function $w(p,q)$ has three components that determine the contribution of pixels q in the neighbourhood $B_\epsilon(p)$ to the inpainting value $I(p)$. The directional component $dir(p,q)$ has pixels q contribute more if the direction of the vector **qp** is close to the direction of the vector normal to the boundary of the inpainting region at p. The vector normal to the boundary is equal to ∇T [1]. The distance component $dst(p,q)$ gives more weight to points q that are near p, so the distance component is inversely proportional to the norm of **qp**. In our implementation, we use Euclidean distance. Finally, the level set component $lev(p,q)$ gives more weight to pixels whose distance maps T are

similar. The equations of the weighting function and its components are given by

$$w(p,q) = dir(p,q) \cdot dst(p,s) \cdot lev(p,q) \tag{3}$$

$$dir(p,q) = \frac{\mathbf{qp} \cdot \nabla T(p)}{||\mathbf{qp}||} \tag{4}$$

$$dst(p,s) = \frac{1}{||\mathbf{qp}||^2} \tag{5}$$

$$lev(p,q) = \frac{1}{1 + |T(p) - T(q)|} \tag{6}$$

2.2 Solve Eikonal equation

The value of the distance map T is the solution to the Eikonal equation. First, we discretize Equation 1 using finite differences, this is given by

$$\max(\Delta^{-x}T, -\Delta^{+x}T, 0)^2 + \max(\Delta^{-y}T, -\Delta^{+y}T, 0)^2 = 1 \tag{7}$$

When evaluated at point $p = (i, j)$, $\Delta^{-x}T(i,j) = T(i,j) - T(i-1,j)$ and $\Delta^{+x}T(i,j) = T(i+1,j) - T(i,j)$. $\Delta^{-y}T$ and $\Delta^{+y}T$ is obtained similarly [1]. The pseudo code to solve Equation 6 is provided in Telea's article and was the basis for our implementation.

Then, as is common in Fast Marching Methods adapted for inpainting, Equation 7 is solved in p's 4 quadrants (where p is considered the origin). The minimum of these 4 solutions is set to be the new value of $T(p)$ [6]. This describes how the T value for BAND pixels is updated, which will determine the order in which pixels are propagated.

3 Results

We present the results obtained from our implementation of the inpainting method. We show how our algorithm performs using different types of masks that are common in applications of inpainting (e.g. restoring thin damaged regions, removing text and unwanted objects). Then, we compare our results with those of OpenCV's implementation of the same method.

As can be seen in Figure 2, the algorithm manages to inpaint pixels in regions of low intensity changes, such as the region between the soccer player's head at the ball. However, there are discrepancies between the source image and output image where edges occur (such as on the soccer player's thumb and ear in Figure 2). There, the image is noisy and there is no clearly defined edge. As we will discuss further, the main weakness of our implementation is its performance in regions of large changes in intensity.

The method was tested with B_ϵ sizes ranging from 2 to 15 ϵ (with ϵ being one pixel). The best results are produced with ϵ in the range of 3 to 8. Figure 3A has ϵ equal to 3, Figure 3B has ϵ equal to 8. The differences between

Figure 2.

those results are significant at edges but do not impact the performance of the algorithm inside homogeneous parts of the image. As we can see in this example, different values of ϵ produce significantly different results which is why we allow users to choose their own value of ϵ. Generally speaking, larger values of ϵ result in more blurring of the inpainting region, which is fine depending on the application. For thicker inpainting regions it becomes necessary to use a larger ϵ in order to match the context of the image.

Figure 3A. (left) and **Figure 3.** (right) is a comparison between results obtained with different values for ϵ.

The following is a visual and numerical comparison between the results obtained with our algorithm and OpenCV's implementation of a FMM inpaint-

ing method. Visually the outputs are similar at points with little changes in intensities. However, at the edges (thumb and ear in Figure 2), the results from openCV are much smoother and have a clearly defined edge. To compare the images quantitatively, we subtracted one image from the other, Figure 4 shows the results of this operation. As expected most of the differences between the two images happen at edges which were to be inpainted. More generally, the differences between the images occur more in regions of high varying pixel intensity. This would explain why there are many differences on the region across his jersey. As there are many different colours it is a region of high variation in pixel intensity which our algorithm failed to smooth over as well as the openCV implementation. In regions of low intensity, our algorithm performs extremely similarly to the openCV implementation. The top region to be inpainted is barely visible in Figure 4, indicating that our results and the openCV results where almost identical.

Figure 4. A numerical comparison of results obtained with different implementations

Visually, the results obtained with OpenCV's algorithm give smoother edges and preserve the shape of objects in the scene better. We can see this highlighted on the ropes of the ring in Figure 5, which look more coherent in OpenCV's results. In our implementation, the ropes of the ring do not form a cohesive object. This issue most likely stems from the same implementation flaw which causes the algorithm to perform poorly at regions of high variation in pixel intensity. This will be discussed more thoroughly in the discussion section.

4 Discussion

Several issues had to be resolved when implementing the pseudo code provided by Telea [1]. Most of the issues arose in the inpainting method which determines the RGB values at a given pixel. The first bug encountered was with

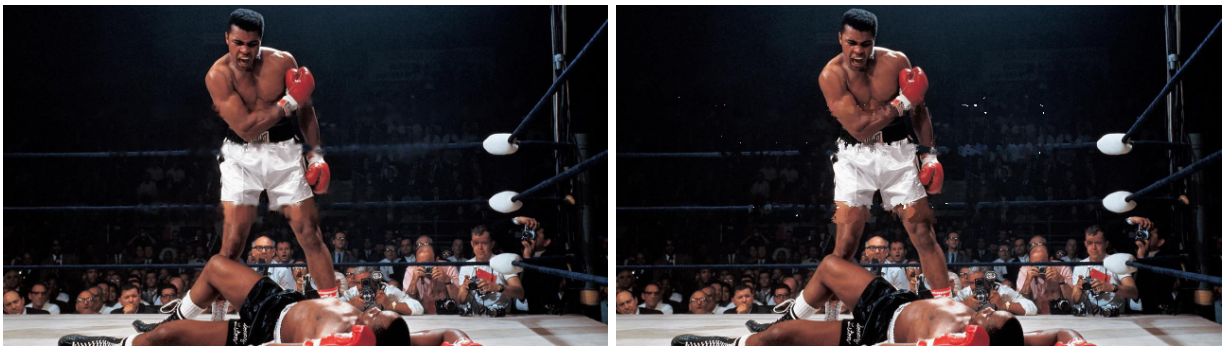

Figure 5. A visual comparison of results obtained with different implementations. OpenCV result on the left, Our result on the right

the weighting function $w(p,q)$ (see Equation 3). Often, we would get negative weighting values. This would result in the algorithm inpainting pixels with negative values. To remedy this we take the absolute value of $w(p,q)$. Open CV's implementation resolves this issue in the same manner. The next issue was that the value of $w(p,q)$ would often be zero. Since calculating the inpainting value involves dividing by $w(p,q)$, it led to zero division errors. This was caused by the dot product of the gradient of the distance map with the vector between q and p. This would cause the direction component of $w(p,q)$ to be zero. In such cases we would set the direction component to a small value (10^{-6}).

As mentioned in the results section our algorithm struggles with smoothness in regions with high variation of pixel intensity which results in discontinuities at edges. In the paper Telea describes the following algorithm to inpaint a specific pixel.

Algorithm 1 Telea's pseudo code

for $all\ \ (k,l)\ \ in\ \ B_\epsilon$ **do**
  $r = vector\ \ from\ \ (i,j)\ \ to\ \ (k,l)$
  $w = dir * dst * lev$
  $\nabla I = (I(k+1,l) * I(k-1,l), I(k,l+1) * I(k,l-1))$
  $Ia+ = w * (I(k,l) + \nabla I * r)$
  $s+ = w$
end for
$I(i,j) = Ia/s$
dir, dst, lev are as defined in equations (4, 5, 6)

The main issue with adapting the pseudo code occurred when calculating the image gradient. Using central differences, it would often happen that the values of the gradient would be very large. This caused the pixel RGB values to be larger than expected or outside the supported range. When experimenting with different ways to resolve this, we tried running Telea's method without using the image gradient. This yielded satisfactory results. All the results

shown in section 3 are achieved with this method. This is likely the main reason the algorithm was not as smooth around the edges. If we were successful in weighing pixel values with the gradient as Telea described, we would get more clearly defined edges and better smoothing in other areas with high variation in pixel intensity. This is also the reason why the ropes in Figure 5 are not as precisely inpainted in our implementation as in OpenCV's implementation. Other attempts to factor in the gradient of the image, such as normalising the gradient and calibrating the inpainting value, failed.

Looking at other implementations such as the original implementation by Telea or OpenCV's implementation, we find different ways of dealing with the image gradient. We will discuss how OpenCV's implementation deals with factoring in the image gradient when inpainting a pixel. It is important to note that the pixel inpainting equation in OpenCV's method is different from the one we implemented (Equation 2).

Algorithm 2 OpenCV pseudo code

for $all\ \ (k,l)\ \ in\ \ B_\epsilon$ **do**
  $r = vector\ \ from\ \ (i,j)\ \ to\ \ (k,l)$
  $w = abs(dir * dst * lev)$
  $\nabla I = (I(k+1,l) * I(k-1,l), I(k,l+1) * I(k,l-1))$
  $Ia+ = w * I(k,l)$
  $Jx- = w * (\nabla I * r.x)$
  $Jy- = w * (\nabla I * r.y)$
  $s+ = w$
end for
$I(i,j) = Ia/s + (Jx + Jy)/(\sqrt{Jx^2 + Jy^2} + 10^{-20}) + 0.5$

OpenCV's method separates the weighting of the neighbouring image pixels and the image gradient into two separate components. And then the value of the actual pixel is defined by Algorithm 2. The constants in the final equation in Figure 6B ensure there is no zero division error and pixel values are greater than zero. We tried adapting our code to use an inpainting function similar to that of OpenCV's, allowing us to factor in the image gradient. However, that did not improve our results. In fact, the results looked almost identical to our original implementation. From this we conclude that there are other differences between the two implementations that lead to distinct outputs. One big difference between the implementations is that OpenCV's method uses three distinct distance maps (one for each colour channel). Our implementation only uses one distance map, so it is possible that. It is likely from this comparison that the reason our algorithm performs worse than OpenCV's is due to the different inpainting technique, and not to the implementation of the FMM. More specifically, these discrepancies are most likely caused by how the gradients of the image and the distance map are treated and weighted in our implementation. .

Some further suggestions that could improve the performance of the algorithm include executing the method recursively. We found that running the

inpainting method recursively improved the method's performance near edges. However, running it more than twice started distorting the inpainting values to colours that did not correspond to the context of the image. Additional testing and fine tuning is necessary to draw a conclusion. Another improvement could be to smooth the output image to decrease noise at the edges.

We also reached out to Professor Alexandru Telea for advice, he suggested different ways in which we could improve the algorithm. The suggestion to consider a circular neighbourhood instead of semi-circular neighbourhood (as outlined in section 2.1) improved the performance of the algorithm. Another suggestion was to add more components to the weighting function WEIGHT, such as factoring in information about contrast. However, this did not lead to better results.

Division of Tasks

After deciding on the topic of our project, we divided up the subjects on which we all had to do research. This included getting familiar with Telea's paper and other papers which describe diffusion-based methods that utilise PDEs to propagate the inpainting region boundary, as well as looking at other implementations of the method. For the coding part, Ade Thornhill and Miles Weberman wrote the first draft of the code and Jonas Vinson did the debugging and ensured the code gave accurate results. For the paper, Ade Thornhill wrote the introduction, Miles Weberman wrote the Method and Jonas Vinson wrote the results. The discussion was written by Jonas and Miles. The group met up regularly to discuss progress and proofread each other's writings.

References

[1] Alexandru Telea (2004) An Image Inpainting Technique Based on the Fast Marching Method, Journal of Graphics Tools, 9:1, 23-34, DOI: 10.1080/10867651.2004.10487596

[2] Elharrouss, O., Almaadeed, N., Al-Maadeed, S. et al. Image Inpainting: A Review . Neural Process Lett 51, 2007–2028 (2020). https://doi.org/10.1007/s11063-019-10163-0

[3] M. Bertalmio, A. L. Bertozzi and G. Sapiro, "Navier-stokes, fluid dynamics, and image and video inpainting," Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001, 2001, pp. I-I, doi: 10.1109/CVPR.2001.990497.

[4] C. Guillemot and O. Le Meur, "Image Inpainting : Overview and Recent Advances," in IEEE Signal Processing Magazine, vol. 31, no. 1, pp. 127-144, Jan. 2014, doi: 10.1109/MSP.2013.2273004.

[5] Chan, T., Shen, J. Non-Texture Inpainting by Curvature-Driven Diffusions (CCD). UCLA CAM TR 00-35, Sept. 2000.

[6] J. A. Sethian. "A Fast Marching Level Set Method for Monotonically Advancing Fronts." Proc. Nat. Acad. Sci. 93:4 (1996), 1591—1595.

Source Code

`https://github.com/MilesWeberman/FMM_image_inpainting`