Fast Marching Method for Outpainting of Certain Gradient Backgrounds

D. Gueorguiev , 11/7/2024

# Introductory Notes

//TODO: explain the history of the FMM, discuss the inpainting modeling work historically, describe the inpainting models used currently and how the one developed in Telea’s paper will be used in modified form to create outpainting model for slowly varying gradient backgrounds.

# Model for Inpainting

Consider the figure below in which one must inpaint the point situated on the boundary of the region to inpaint .

We take a neighborhood with radius of the known image around . The inpainting of should be determined by the values of the known image points close to (i.e. in ). Grayscale images are considered in this model , color images are handled as an obvious extension by the presented model and algorithm.

boundary

region to be

inpainted

known

neighborhood

known image

**a)**

**b)**

When is small enough a first order approximation of the image in point , given the image and the gradient values of point :

(1)

Next, we inpaint point as a function of all points in by summing the estimates of all points , weighted by a normalized weighting function :

(2)

The weighting function is designed in such way that the inpainting of propagates the gray value as well as the sharp details of the image over .

In [1] , is determined based on three components – a *directional* component , *geometric distance* component , and the *level set* distance component .

(3)

where

(4)

(5)

(6)

The *directional component* ensures that the contribution of the pixels close to the normal direction (which is the information propagation direction) is higher than for those farther from .

The *geometric distance* component decreases the contribution of pixels geometrically farther from .

The *level set* distance component ensures that pixels close to the contour through p contribute more than farther pixels.

Overall, the above factors model the manual inpainting heuristics that describe how to paint a point by strokes bringing color from a small region around it.

For up to about six pixels i.e. when inpainting thin regions, and have a weak effect. For thicker regions to inpaint we could use an of 12 pixels and using and provides better results than using alone. Inpainting is the best visually when all three components have been used.

A black oval with a white square

Description automatically generated with medium confidence

Figure: thick region to inpaint (a) and result (b). Effect of the weighting functions: direction (c), direction and geometric distance (d) , direction and level set distance (f). The figure is taken from [1].

## Adding Fast Marching Method to the Inpainting Process

Eq. (2) gives the rule how to inpaint a point on the unknown’s region boundary as a function of known image pixels only.

To inpaint the whole , we iteratively apply Eq. (2) to all the discrete pixels of , in increasing distance from ’s initial position , and advance the boundary inside until the whole region has been inpainted. Inpainting points in increasing distance order from ensures that areas closest to known image points are filled in first, mimicking manual inpainting techniques.

Implementing the above requires a method that propagates into by advancing the pixels of in order of their distance to the initial boundary . For this we use the Fast-Marching Method (FMM). FMM is an algorithm that solves the Eikonal equation.

## The Eikonal equation

The Eikonal equation is given with

on , with on (7)

The solution of Eq. (7) is the distance map of the pixels in to the boundary . The level sets (or *isolines*) of are exactly the successive boundaries of the shrinking that need to be inpainted.

## The FMM algorithm

{

}

Excerpt 1: the FMM algorithm discussed in Telea’s paper

The FMM algorithm guarantees that the pixels of are always processed in increased order of their distance-to-boundary . Thus the algorithm will inpaint the closest pixels to the known image area first. The FMM algorithm explicitly defines and maintains a *narrow band* that separates the known from the unknown image *and* specifies which is the next pixel to inpaint. The narrow band can be implemented as *min heap*. For details and Python code see the appendix.

# The Outpainting Problem

A diagram of different colored lines

Description automatically generated

Figure: Solution of the Eikonal equation with open initial contour (depicted with black). Marching contours produced with [scikit-fmm](https://github.com/scikit-fmm/scikit-fmm).

on

Figure: Outpainting scenario which is not simple but is realistic

How can we solve the outpainting problem in a realistic way?

# References

[1] [An Image Inpainting Technique Based on the Fast Marching Method, Alexandru Telea, 2004](https://github.com/dimitarpg13/image_processing/blob/main/literature/articles/inpainting_algorithms/An_Image_Inpainting_Technique_Based_on_the_Fast_Marching_Method_2004JGraphToolsTelea.pdf)

[2] [A Fast Marching Level Set Method for Monotonically Advancing Fronts, J.A. Sethian, 1996](https://github.com/dimitarpg13/image_processing/blob/main/literature/articles/inpainting_algorithms/sethian-1996-a-fast-marching-level-set-method-for-monotonically-advancing-fronts.pdf)

[3] [Image Inpainting, M. Bertalmio, G. Shapiro, V. Caselles, C. Ballester, 1999](https://github.com/dimitarpg13/image_processing/blob/main/literature/articles/inpainting_algorithms/Image_Inpainting_bertalmio_1999.pdf)

[4] [scikit-fmm: the fast marching method for Python](https://github.com/scikit-fmm/scikit-fmm)

[5] [pyheal: Fast Marching Inpainting, minimalistic Python implementation](https://github.com/olvb/pyheal)

# Appendix

## Code implementation of the algorithm discussed in Telea’s paper

import heapq

from enum import Enum

class PixelState(Enum):

KNOWN = 1

INSIDE = 2

BAND = 3

def inpaint\_region():

{

# STEP 1: extract P(i,j) = top of the narrow\_band heap

#//**TODO:** write this code

flag[i,j] = PixelState.KNOWN

for (k,l) in [(i-1,j), (i,j-1), (i+1,j), (i,j+1)]:

if flag[k,l] != PixelState.KNOWN:

if flag[k,l] == PixelState.INSIDE:

# STEP 2:

flag[k,l] = PixelState.BAND

# STEP 3:

inpaint\_pixel(k,l)

# STEP 4:

dist\_to\_boundary[k, l] = min(solve\_eikonal(k-1,l,k,l-1),

solve\_eikonal(k+1,l,k,l-1),

solve\_eikonal(k-1,l,k,l+1),

solve\_eikonal(k+1,l,k,l+1))

# STEP 5: insert pixel (k,l) in narrow\_band heap

#//**TODO:** write this code

}

def solve\_eikonal(