Solving Hole Filling Exercise

Matanel Sindilevich

Table of Contents

[Abstract 2](#_Toc449223569)

[Tasks 2](#_Toc449223570)

[Task 1: Find the boundary of the hole 2](#_Toc449223571)

[Task 2: Color missing pixels using interpolation 2](#_Toc449223572)

[Task 3: Show original and processed images 2](#_Toc449223573)

[Task 4: Assess naïve implementation of 2 in terms of 3](#_Toc449223574)

[Task 5: Approximate implementation of 2 with 3](#_Toc449223575)

[Task 6: Bonus — Exact color interpolation method with 3](#_Toc449223576)

[The Application 4](#_Toc449223577)

[Obtaining source code and compilation 4](#_Toc449223578)

[Source code 4](#_Toc449223579)

[Compile and run 4](#_Toc449223580)

[Tests 4](#_Toc449223581)

[Usage 4](#_Toc449223582)

[Open a Grayscale Image 4](#_Toc449223583)

[Fill Holes 6](#_Toc449223584)

[Possible improvements 7](#_Toc449223585)

[Software design 8](#_Toc449223586)

[Algorithms 8](#_Toc449223587)

# Abstract

The goal of the Hole Filling exercise is solving a problem of filling in data into a region, based on interpolating data, available on the boundary of that region, in terms of Computer Vision[[1]](#footnote-1).

Given a grayscale image, where each pixel value is a float in the range , and invalid (missing) values which are marked with the value form a single hole in the image.

1. Find the boundary of the hole.
2. Given an image , the boundary pixel locations , a missing pixel’s location , and a weighting function set the value of pixel as follows ( 1 )
3. Show the input image, the image with its boundary marked and the filled image.
4. If there are boundary pixels and pixels inside the hole, what’s the complexity of the algorithm from 2?
5. Write an algorithm that approximates the result from ‎2 in to a high degree of accuracy.
6. Bonus — Write an algorithm that finds the exact solution in .

# Tasks

Below is the breakdown of tasks and their solutions. Technical assumption was made as to the way to encode missing pixels in standard 8bpp grayscale BMP images. It is said a pixel is invalid if the source color value of the pixel in the BMP is white (). Hence it imposes using grayscale images, where any pixel’s laying outside the hole color value is in range . Sample images are part of the Computer Vision open project[[2]](#footnote-2).

## Task 1: Find the boundary of the hole

Given an image, a 2D surface with the width of and the height of . A simple iterative algorithm is used to find pixels that belong to the hole: . For each found, the algorithm searches its closest neighbors for possible boundary pixels , using a pixel pattern, and stores them in a set, to eliminate duplicates. Since the exercise assumes there is only a single hole in the image, the algorithm exits once a row with no hole pixels was found, in case it was immediately preceded by a row with hole pixel(s).

## Task 2: Color missing pixels using interpolation

A naïve approach to solving the problem is iterating through the set of the invalid pixels , calculating color value for as the result of running ( 1 ) over the set of all the boundary pixels .

## Task 3: Show original and processed images

Using Emgu.CV[[3]](#footnote-3) to display images.

## Task 4: Assess naïve implementation of 2 in terms of

If there are boundary pixels and pixels inside the hole, the naïve approach (see Task 2: Color missing pixels using interpolation) yields . However, from the geometrical point of view, is the length of hole’s perimeter, whereas is the hole’s square. Thus when grows, grows as well and its growth depends on that of . Therefore, it is possible to express the complexity as .

## Task 5: Approximate implementation of 2 with

There are several strategies to interpolate an approximate value of a pixel. The application implements the color interpolation on fixed-sized random boundary subset approach.

The algorithm operates with boundary pixels and hole’s pixels.

As hypothesized in Task 4: Assess naïve implementation of 2 in terms of , the naïve implementation of ( 1 ) yields . The reason is that pixels of the hole boundary depends on the number of hole’s pixels .

The approximate algorithm is based on the idea of constraining the number of boundary pixels, used during color interpolation, to a fixed number of elements. Therefore, the color sampling subset of boundary pixels is fixed and does not depend on , making the approximate implementation .

Given is a fixed-size subset of .

The implemented approach is interpolating color of based on . The boundary pixels that compose the fixed-size subset are randomly selected for every . The idea is that since color of solely depends on , a carefully selected number of evenly-distributed boundary pixels gives a good approximating of the color; while keeping the complexity at due to the fixed-size nature of .

## Task 6: Bonus — Exact color interpolation method with

Whilst not implemented in the application, below is theoretical explanation of an exact color interpolation method with complexity of .

The idea is that interpolating color for an invalid pixel need not take all the boundary pixels into account. Instead, only the nearest neighbors are allowed to participate in pixel’s color interpolation. Indeed, the weighting function in ( 1 ) is inversely proportional to distance between the invalid pixel under question and a boundary pixel. Applying the k-Nearest Neighbors algorithm in conjunction with a kd-tree, introduces searching the nearest neighbors for every of the pixels inside the hole, in a kd-tree. Since kd-tree search is , the resulting complexity is .

The algorithm is known as a modified version of Shepard’s method. First proposed by Robert J. Renka (“Multivariate Interpolation of Large Sets of Scattered Data”, 1988), modified interpolant have following form:

The modified Shepard’s method has the following aspects:

1. Only points are used for interpolation — a set of nearest neighbors of , which is denoted by .
2. Instead of constant there is nodal function .
3. kd-trees are used for efficient k-NN search with complexity.
4. Locality of interpolation algorithm: depends only on nearest neighbors of which significantly improves quality of interpolation.

# The Application

Solving the exercise is implemented as a 64-bit WinForms[[4]](#footnote-4) application, compiled with Visual Studio 2015 Update 2, and running on .Net Framework 4.6. The application depends on Emgu.CV v3, a cross platform .Net wrapper to the OpenCV image processing library, for image I/O and the matrix addition and multiplication operations. Another dependency is log4net[[5]](#footnote-5) v2 for logging. The application is part of the Computer Version open project. The application was designed with maintainability and extensibility in mind, including unit-testing to ensure domain-specific behavior.

## Obtaining source code and compilation

The application is available in form of C# source code. It was tested to compile and run on a Windows 10 x64 machine, with .Net Framework 4.6 and Visual Studio 2015 Update 2 installed. The x64 architecture and .Net Framework 4.6 are prerequisites.

### Source code

Clone the repository with Git: git clone <https://github.com/sindilevich/ComputerVision>.

### Compile and run

Open the HoleFilling/HoleFilling.sln file in Visual Studio 2015 Update 2, on a Windows x64 machine with .Net Framework 4.6 installed. Make sure the HoleFilling project is set as the default project. Hit F5 to run the application in the debugger, no compile errors are expected at this stage.

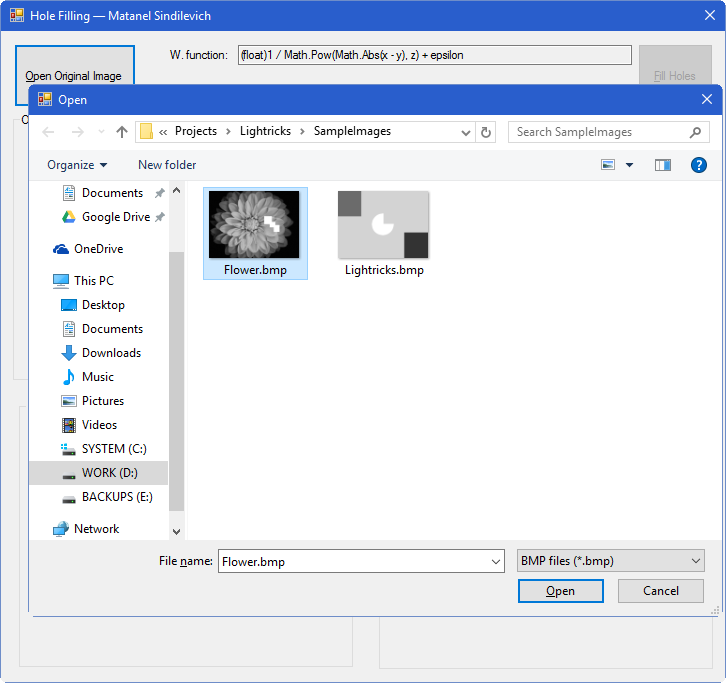
### Tests

Running unit-tests is available through Visual Studio.

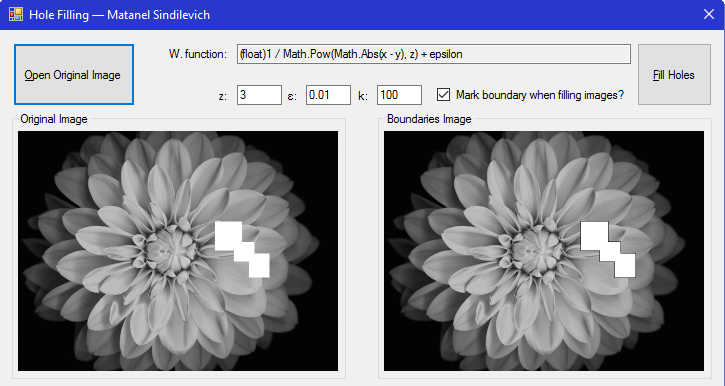
## Usage

### Open a Grayscale Image

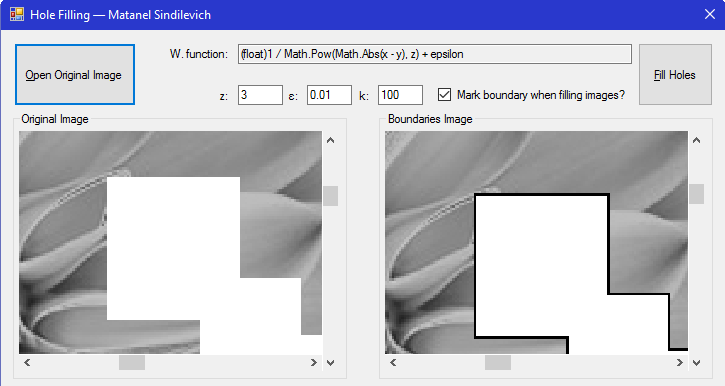
Start by opening a grayscale image in the application by hitting the Open Original Image button. The application includes a set of grayscale BMP images with holes, denoted by white regions.



Once a grayscale image is selected and loaded from disk, it is displayed in the Original Image quadrant of the window. The application automatically finds the boundary of the hole in an image and marks it with a one-pixel black line, and populates the Boundaries Image quadrant with the relevant visual data.

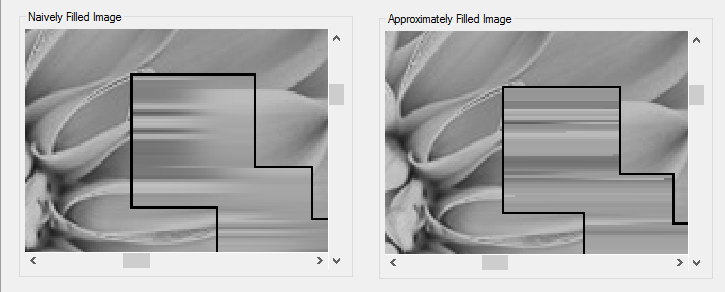


It is possible to pan and zoom into the images for finer details within each quadrant.



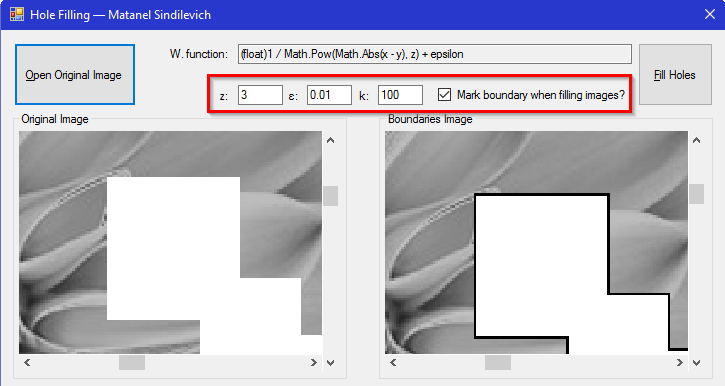
### Fill Holes

Hitting the Fill Holes button starts a chain of computations to fill pixels within the white, data-missing region with interpolated data from the region’s boundary. See ‎2 and ‎5 above for details. The Naively Filled Image quadrant represents a naïve implementation of color interpolation, whilst the Approximately Filled Image quadrant is characterized by a faster, yet approximate calculation of color interpolation.



The end result of the holes filling process is controlled by , , and . Where and are part of the weighting function as defined in ‎2. — parameter to the approximate color interpolation algorithm as defined in Task 5: Approximate implementation of 2 with . And Mark boundary when filling images? aids in user experience when examining the filled images.

Please note, changing the values of , , and Mark boundary when filling images? takes effect only at subsequent hitting the Fill Holes button.



# Possible improvements

There are several ways to improve the application in terms of both software design and algorithms. Below are some of the major points to consider for further improvement.

## Software design

1. As application grows larger, introducing a MV-\* pattern becomes of a great necessity.
2. Although the code is generally based on Dependency Injection principle, it still lacks an automatic IoC container.

## Algorithms

Whilst the application offers faster algorithms for hole filling, it can still benefit from parallelizing tasks. Parallel algorithms may run faster on a multiprocessor/multithreaded machine, even considering the penalty of increasing data management and synchronization complexity.

1. Original definition of the exercise — https://github.com/sindilevich/ComputerVision/blob/master/hole\_filling.pdf [↑](#footnote-ref-1)
2. https://github.com/sindilevich/ComputerVision [↑](#footnote-ref-2)
3. http://www.emgu.com/wiki/index.php/Main\_Page [↑](#footnote-ref-3)
4. WinForms was chosen as basis for rapid development. In commercial applications, targeting the Windows desktop platform, a WPF or UWP application is recommended. [↑](#footnote-ref-4)
5. http://logging.apache.org/log4net [↑](#footnote-ref-5)