University of Illinois

Final Report

Image Inpainting

Hongyi Li
Tejas Jayashankar

May 9, 2018

Contents

1 Abstract

In this project we will study four different methods to perform image inpainting. We will specifically concentrate on techniques that involve the solution of Partial Differential Equations (PDE) and convolutional methods. We will also comment on anisotropic and isotropic diffusion and the use of these methods in PDE based image inpainting. We will tabulate results and compare the methods against each other to characterize the advantages and disadvantages of each method.

2 Introduction

Images contain a wealth of important information Unfortunately, its usually costly or even impossible to reconstruct images after they have been contaminated or damaged. For example, images tend to lose their contrast and develop scratches over long periods of time. Often such images might contain historical information or information necessary in the solution of a problem.

Image inpainting was initially used to restore scratched frames on films. As time progressed and after invention of the computer the technique of image inpainting has been expanded to fill in digital images, alter the scenes and textures of images and even in the removal of objects. The art of restoration of pictures has been around for a long time. Digital image inpainting on the other hand started to boom in the 20th century. Since the invention of the computer, people started to experiment with filters such as the median filter to perform inpainting. Since then there have a number of methods based on PDEs and linear and non-linear filters.

2.1 Project Contents

Nowadays, engineers usually divide image inpainting algorithms into two categories. One is convolution-based, while the other is PDE-based. In our project, we have 3 convolution-based methods. One is the median filtering, one is based on 2D filtering around damaged areas using an averaging filter and the last one is based on gradients of pixel values in the surrounding area of the damaged area so that we can estimate values at those damaged pixels. In addition, we have two PDE-based methods, which include anisotropic diffusion and isotropic diffusion. These two methods simulate heat diffusion phenomenon and diffuse the surrounding information into the damaged areas. Overall, PDE-based methods utilize more complicated math and thus yield more delicate and accurate results. However, solving the differential equations also require a large amount of running time. In comparison, convolution-based methods use simpler mathematics and thus yield less satisfactory results. However, it is better in running time. We can see the advantages and disadvantages of these methods more clearly in the following sections. We chose the above methods so that we could understand the need for inpainting based on PDE and analyze and determine the appropriate situation for using a convolutional or PDE based inpainting method.

3 Method 1: Median Filtering

Awati and Patel[1] describe a simple method for image inpainting based in median filtering. The assumption made is that the damaged pixels usually have extreme values, either 0 (extremely dark) or 255 (extremely bright). In addition, a rectangular search area is defined, which contains all damaged pixels. Potentially, it may also contain uncontaminated pixels. The general algorithm as described in [1] is as follows:

1: **for** each pixel in the search area **do**
2: **if** the pixel value is 0 or 255 **then**
3: Define a 41x41 mask around the pixel, with the damaged pixel in the center.
4: Order the 41x41 pixel values in the mask in ascending order.
5: Replace the damaged pixel with the medin value in the mask.
6: **end if**
7: **end for**

When we encounter an extreme pixel value like 0 or 255, its very likely to be contaminated.

Results can be found at the end of the document. Results inlcude test images and running times of the algorithm.

3.1 Implementation Details

We implement this algorithm in C++. Details about the scripts can be found in Appendix A along with the commands for compiling the code. We utilize the image classes used for previous labs so that we can easily read in input images and save results in PNG files. We use a vector to store all pixel values in masks around damaged pixels and use sort function in the algorithm class to order these pixel values. To replace the damaged pixel, we just equate it to the middle value in the sorted vector. We will input the coordinates of the top left corner and the bottom right corner. Then we use a double for loop to go through the searching area in the raster-scan order.

3.2 Method Evaluation

First of all, we can see that the median filtering method is very easy to implement. As we have described, C++ has all the necessary data structures and sorting algorithms for this method. Implemented in such a low-level language, it also behaves pretty well in running time. As we can see, all of the test cases can finish within a quarter of one minute. For those images with small damaged and search areas , like lena (0-4), the program can even finish within a second.

Basically, this algorithm will go through the search area and process all damaged pixels. Lets say there are M contaminated pixels in total. The time to process each damaged pixel, including collecting mask pixel values, sorting and picking up the median, is constant. Therefore, we can estimate the total running time to be O(M).

Second, the visual quality is pretty good for damaged pixels in a uniform background. For example, after processing lena0.png or lena4.png, the small square is gone and completely merged into the white backgrounds in the face or hat. Only by looking at the image very close to the screen can we perceive the previous damaging effects.

Third, the method isnt very good at recovering texture and is very likely to produce shading effects. For example, in the test cases of lena1.png and lena2.png, the damaged square is on Lenas hair. However, after processing, we see a square that has a color closer to the surrounding area. Of course, the visual quality is slightly better, but it obviously doesnt fit into the texture around the damaged area, which is pretty annoying if we look at the picture carefully. This is because this method can only approximate the damaged pixel value by taking the median in the mask and cannot predict more precisely according to information from neighboring pixels. In addition, as we replace more and more damaged pixel values, certain values will dominate in masks for later damaged pixels. As a result, the damaged pixels we process later tend to have values that we produce and replace earlier. For instance, in the test case of letter.png, the shading effect is so strong that we have a large MSE value and the visual quality is still not acceptable even after restoration.

Finally, because of the reasons we just mentioned, this method can sometimes shrink or enlarge original image components. In the raster-scan order, we usually process damaged pixels in the top left corner first. Then the damaged pixels in the bottom right corner tend to have values we replace earlier in the top left corner. As a result, this may make some objects in the original images lack certain corners or edges, especially when the damaged areas cross certain edges or corners. In the test case of lena3.png, Lenas hat loses a corner at the edge after restoration.

4 Method 2: Convolution Based on Gradient

The general idea of this method is to analyze the difference between the damaged pixels and their neighboring good pixels[2]. Then, according to these differences, we assign different weights to the neighboring good pixels. The damaged pixel value is synthesized by the weighted neighboring good pixels. We do this for each damaged pixel and we keep doing this for the whole image for certain times so that the MSE is small enough and the visual quality is acceptable.

To achieve this, the specific procedure is as follows. In general, we divide all pixels into three categories. The first category includes pixels that are not contaminated. We call these pixels as undamaged pixels. The second category includes contaminated pixels whose values arent replaced yet. We call these pixels as damaged pixels. The third category includes contaminated pixels whose values are already replaced with synthesized values from the neighbourhoods. We call these pixels as restored pixels. For each damaged pixel, we define a 3x3 region around it. That is to say, the damaged pixel is the center of the 3x3 area. Then, for each good or restored pixel in the neighborhood, we calculate the difference between the pixel (P_k) and the damaged pixel (P).

Lets denote the difference as x_k. Then, (the function f will return the pixel value for a certain pixel.) $x_k = f(P_k) - f(P)$.

With these difference values for these good or restored pixels in the neighborhood, we can continue to calculate the weights of these pixels. The equation is: $w(k) = \frac{1}{n}F(x_k)$, where n represents the total number of good or restored pixels in the 3x3 neighborhood. The function F is a function that assigns weights according to the pixel difference. Its equation is as follows[2]:

$$F(x) = \begin{cases} 1 - \frac{x^2}{\alpha^2} & \text{if } |x| \leq \frac{\alpha}{2} \\ (\frac{x}{\alpha} - 1)^2 & \text{if } \frac{\alpha}{2} \leq |x| \leq \alpha \\ 0 & \text{if } |x| \geq \alpha \end{cases} \tag{1}$$

As we can see, this function in general assigns more weights to the neighboring pixels that have closer values to the central pixel. As a result, the damaged pixel value can become closer and closer to the good pixel values, under repetitive iterations. α is a user-defined constant that is used as the threshold value for pixel difference. By using different alpha values, we can thus allow different smoothness. If the difference is too large, we shouldnt consider it when synthesizing final restored pixel values at all, as reflected by the third line in the equation. In our program, we use 255.0 as α. In this way, each good or restored pixel will be given some weight, although pixels that are too different from the damaged pixel will be weighted slightly in the final restored value.

Finally, we replace the central damaged pixel with: (The function f' represents the new value of a pixel.)[2]

$$f'(p) = (1 - \sum_{k=1}^{n} w(k))f(p) + \sum_{k=1}^{n} w(k)f(k) \tag{2}$$

After replacing, we mark the damaged pixel as RESTORED. Then we continue this process for each damaged pixel.

After we process all damaged pixels, we mark all resoted pixels as DAMAGED again and repeat the above process. We keep doing this iteration for a certain times until the image quality is acceptable.

The pseudocode is as follows:

```
for i = 0 to 100 do
    for each damaged pixel p do
        f'(p) = (1 − Σ_{k=1}^{n} w(k))f(p) + Σ_{k=1}^{n} w(k)f(k)
        Mark the pixel as RESTORED
    end for
    Mark all RESTORED pixels as DAMAGED
end for
```

Results can be found at the end of the document. Results include the test images and the running times of the algorithm.

4.1 Implementation Details

We implement this algorithm in C++, whose compiling and folder instructions are specified in Appendix A. Whenever we run the program, we will input the coordinates of the start searching point (x, y) and a searching range r. Then we will go through the rectangular area with (x,y) and (x+r, y+r) as corners in the raster-scan order. We assume that this area contains all contaminated pixels. We will process pixels whose values are 0 or 255. Like Method 1, these pixel values are very likely to happen on damaged pixels.

We use a 2D array whose dimensions are the height and width of the input image to record whether each pixel is undamaged, damaged or restored. We have several helper functions that need these information and we pass the 2D array pointer to these helper functions.

4.2 Method Evaluation

This algorithm also processes damaged pixels one by one. Therefore, the run-time is linear with the size of the damaged region.

First, we can see this algorithm is pretty fast, as well. For example, processing letter.png takes the median filtering method the most time – 14.78 seconds. However, it only takes this method 3.37 seconds. This is because this method only includes common algebraic calculations, which can be finished quickly in low-level hardware. In contrast, the median filtering method contains complicated sorting processes, which can take longer time to finish.

Second, the gradient filtering method performs better in the smoothness of output images, although it has larger MSE values for some test cases. For example, in the output image of letter.png, we can see the high-texture hair pretty clearly. However, in the output image of the media filtering method, the hair is severely affected by the shading remaining letters. In the test case of letter.png, it can even partially recover Lenas eyes, which have lots of details in the original graph. This is because the gradient filtering method focuses more on the relations among neighboring pixels and takes these into account when synthesizing damaged pixel values. Hence, the replaced values can better fit into the overall background, which gives people better visual quality. This is also why the gradient filtering method has fewer shading effects.

5 Method 3: PDE based Inpainting

Bertalmio et al.[3] proposed a technique to inpaint images based on propagating information along the isophotes direction at the boundary. Information is not propagated along the normal direction since such a technique is not successful in maintaining the continuity of slant lines and curves.

The algorithm performs the following update on the image to fill the subset of the image to be inpainted[3], Ω.

$$I^{n+1}(i,j) = I^n(i,j) + \Delta t I_t^n(i,j), \forall (i,j) \in \Omega$$

where

$$I_t^n(i,j) = \delta\vec{L}^n(i,j) \cdot \vec{N}(i,j,n)$$

Here, L^n is the laplacian of the image and it gives an estimate of the smoothness change in the image. Thus, if we propagate in the tangential direction at the boundary, which is given by the vector $\vec{N}$ and by an amount proportional to the change in smoothness, we can continue the isophotes at the boundary. The algorithm was implemented as suggested in the paper.

The algorithm performs a few steps of inpainting followed by a few steps of diffusion. The goal of diffusion is to smoothen the estimated values and diffuse the values inwards into the inpainted region. The diffusion equation proposed by the paper[3] is:

$$\frac{\partial I}{\partial t}(x,y,t) = c(x,y,t)\kappa(x,y,t)|\nabla I(x,y,t)|, \forall (x,y) \in \Omega$$

Here κ is the curvature of the isophotes in the image. We implemented two forms of diffusion: anisotropic diffusion and isotropic diffusion. c is a constant function for isotropic diffusion.

5.1 Anisotropic Diffusion

The implemented anisotropic diffusion method was based on the algorithm proposed by Perona and Malik[5]. We did not perform a discretization of the equation proposed in the paper by Bertalmio et al[3]. The anisotropic diffusion equation proposed by Perona-Malik is[5]:

$$I_t = div(c(x,y,t)\nabla I) = c(x,y,t)\Delta I + \nabla c \cdot \nabla I$$

In our algorithm $c(x,y,t) = \dfrac{1}{1 + \dfrac{||\nabla I||}{K}}$, where K is a dilation constant chosen by the user. We implement the same discrete implementation of the above equation that was proposed in the paper.

$$I_{i,j}^{t+1} = I_{i,j}^t + \lambda(c_N \cdot \nabla_N I + c_S \cdot \nabla_S I + c_E \cdot \nabla_E I + c_W \cdot \nabla_W I)$$

where the subscripts indicate taking the gradient in the North, South, East and West directions. The function c is also evaluated using the corresponding gradient values.

5.2 Isotropic Diffusion

The isotropic diffusion equation is the same as the anisotropic diffusion equation with the function c now a constant. We can discretize the diffusion equation proposed in the paper by Bertalmio et al. to find an expression for the isotropic diffusion equation. Without loss of generality we can set c to be 1.

Hence,

$$\frac{\partial I}{\partial t}(x,y,t) = \kappa(x,y,t)|\nabla I(x,y,t)|, \forall (x,y) \in \Omega$$

The gradient of the curve is the normal to the curve and we can call this $\vec{w} = \dfrac{\nabla I}{||\nabla I||}$. The isophote direction is perpendicular to the normal and we can call this $\vec{v}$. The curvature is calculated using the known formula

for curvature of the isophote $\kappa = \frac{I_{vv}}{I_w}$, where I_{vv} is the hessian of I in the direction of $\vec{v}$ and I_w is the directional derivative of the image in the direction w. Thus, the curvature can be calculated easily using these known vectors and the hessian. On expanding the various equations we get:

$$I_{vv} = \frac{I_y^2 I_{xx} - 2I_x I_y I_{xy} + I_x^2 I_{yy}}{I_x^2 + I_y^2}$$

$$I_w = \sqrt{I_x^2 + I_y^2}$$

$$||\nabla I|| = \sqrt{I_x^2 + I_y^2}$$

Substituting these back, we get a discrete equation:

$$I_{i,j}^{t+1} = I_{i,j}^t + \Delta t I_{vv}$$

Results can be found at the end of the document. Results include test images. The running time of the algorithm was around 3 minutes for each of the test images.

5.3 Implementation Details

The whole algorithm was implemented in MATLAB. As already discussed, most of the code was written just following the suggested algorithm as detailed in the papers.

All image gradients were calculated easily by applying 1D or 2D filters to the image. Central difference was taken in all cases to calculate the gradient. Further implementation details on anisotropic diffusion is detailed in the paper by Perona-Malik. Details about the running the code can be found in the Appendix.

5.4 Method Evaluation

Firstly, we noticed that the results did not vary to a significant precision when Δt was increased. We changed the value of K and observed the results after diffusion. The inpainting method remains the same.

In all cases we observed that anisotropic diffusion yielded better results than isotropic diffusion.

First we tested the code on the Lena image with a patch on her hand. We will show the results for K = 2 and K = 10 in Figure 1. There was not much improvement when increasing K, and hence in the results listed later on, we will provide results for the running 150 iterations of the code which includes 50 rounds of inpainting and 20 rounds of diffusion per iteration. We set K = 10 for anisotropic diffusion. Increasing the value of K, did not improve results in any test case. We use 150 iterations for isotropic diffusion.The MSE and PSNR were calculated for each image. The MSE is calculated between the original images after unergoing a round of diffusion and the inpainted image. This is because each image undergoes a round of anisotropic diffusion as a pre-processing step to smoothen the image slightly.

From Figure 15, we see that the anisotropic diffusion does a very good job at filling in the corrupted region. The MSE in both cases, K = 2 and K = 10, are about the same. The isotropic diffusion also does a nearly clear job, but it still leaves a dark patch at the center of the arm. The MSE is higher in this case.

In Figure 16, we create white streaks on the cameraman image and inpaint this region using a mask that covers the streaks. The anisotropic and isotropic diffusion perform exactly the same on this image, and they result in low MSE. Thus we observe that as the corrupted region get narrower, we can perform better inpainting.

Next, in Figure 17, we perform inpainting on the Lena image corrupted by the letter E. The anisotropic diffusion perform quite well except along the hair. The isotropic diffusion does not perform as well. Thus, we can conclude that isotropic diffusion performs better on smaller and narrower regions. The anisotropic diffusion on the other hand, is able to diffuse values along the inpainted region better.

In Figure 18, we perform inpainted on the Lena image which is heavily damaged by text. The isotropic diffusion performs an unsatisfactory job at removing the text. The anisotropic diffusion reduces the dark spots but still leaves the image heavily damaged and blurred. Thus, we can conclude that the diffusion processes are not very good at removing cluttered corrupted regions. In this case, the test was also thick. We notice that the anisotropic diffusion nearly removes the text in the background of Lena. The diffusion process is not well suited for inpainting along edges and hence we notice corruption when looking at the edges in the background. Lena's lips and chin are much more visible using anisotropic diffusion. Her eye is cut out in both cases.

6 Method 4: Diffusion Filters

We follow the method proposed by Oliveira et al.[4] that makes use of diffusion kernels to perform digital inpainting. This method is useful if the region is not too large and the surrounding area is smooth. Since, the only operation performed is convolution it is a very fast method.

From the results using isotropic diffusion we notice that the diffusion looks very similar to averaging. We see some sort of blurring effect which is a characteristic of a blurring kernel. Thus Oliveira et al. propose that the diffusion kernels should be averaging kernels that have zero weight at the center and average the neighboring 8 pixels based on particular weights.

The two averaging kernels used are[4]:

$$\text{Kernel 1} = \begin{bmatrix} 0.073235 & 0.176765 & 0.073235 \\ 0.176765 & 0 & 0.176765 \\ 0.073235 & 0.176765 & 0.073235 \end{bmatrix}$$

$$\text{Kernel 2} = \begin{bmatrix} 0.125 & 0.125 & 0.125 \\ 0.125 & 0. & 0.125 \\ 0.125 & 0.125 & 0.125 \end{bmatrix}$$

6.1 Implementation Details

The algorithm only requires the image and the mask. The pixels values within the mask are set to 0 and the whole image undergoes convolution with the selected kernel. The pixels values that were filled within the mask are added back to the original image to fill in the inpainted region.

The image is inpainted starting from the border of the region to be inpainted. Thus in each iteration will have 2 or 3 neighboring pixels to each pixel on the border inpainted. The inpainted is performed until the whole region has been filled.

An important normalization step was performed by us, which was not mentioned in the paper. Since not all pixel values are valid/filled when inpainting the region, we divide each new pixel value after convolution by the sum of the coefficients in the kernel that was used. This, step is important to account for the fact that not all neighboring pixels were used for averaging. For example if the neighborhood of the pixel under consideration only had valid/filled pixel in the top row, then this pixel would estimated by the sum of these pixels weights according to the kernel divided by the sum of the top three coefficients of the kernel. In the case where all neighboring pixel values were used, we divide by 1 since the sum of the coeffcicients in each kernel is 1.

At the end of each iteration the mask is updated to account for the pixels that were filled in and to propagate the border inwards. The mask contains a 1 for pixels that are known or filled.

PseudoCode(I, mask)

1: Convolve image with kernel (I_{conv})
2: Convolve mask with kernel (I_{mask}) to obtain sum of coefficients that each pixel uses.
3: Find the new indices that were inpainted (ind)
4: For all i in ind, $I(i) = I(i) + I_{conv}(i)/I_{mask}(i)$
5: Update mask. Set all filled pixels to 1.

Results can be found at the end of the document. Results include the test images. The running times was around 0.1 seconds for each of the test images.

6.2 Method Evaluation

In Figure 19, we again performing inpainting on the Lena image with a patch on her arm. When using either kernel, there appears a bright spot at the center of the region. This is not something we noticed in the diffusion processes based on PDEs. This spot is probably due to the slight change in intensity values along her arm from left to right. This also supports the claim by the authors that this algorithm does not perform very well on larger regions.

We see the robustness of this algorithm from the results in Figure 19. The algorithm completely eliminates

the streaks, without any smudge and the MSE values are very small. If one looks closely at the legs of the tripod in images (c), we can see the larger averaging around the neighborhood of the streak.

From Figure 20, we see that this convolution method also perform very well in removing the E from the image. It also leaves a smaller residual mark than anisotropic diffusion.

This algorithm performs extremely well on the image corrupted by text. Firstly, from Figure 21, this diffusion process preserves the edges much more than diffusion. Lena's lips and eyes are now clearly visible despite being slightly blurred. The MSE values are also much more lower.

Thus, overall, this algorithm performs better on inpainting regions that are narrower and smaller (in the case of text each word was small). Since, we keep on propagating the borders inwards, the estimate becomes better in each iteration. However, as we saw in Figure 18, for larger smooth areas, diffusion performs better.

7 Comparison of Methods

We will now compare the different methods applied to the scenario above. In the above cameraman image we aim to remove the building in distance by applying a mask around the area and applying each of the inpainting algorithms to fill in the region. This is an important application of inpainting other than reconstructing damaged areas. Often inpainting is used to clean images and remove unnecessary objects in the vicinity of the area of interest.

By a quick glance, we notice that all of the algorithms perform a fairly good job in removing the building. On stronger observation of the images, we can see that not all methods fill in the region uniformly. The convolution based on gradient method actually leaves a shaded outline of the building since there is a sharp change in the image gradient on the edges of the building. The median filtering performs quite well, but the filled in area is over smoothed and we actually have a slab of constant intensity in place of the building.

From an aesthetic point of view, the anisotropic diffusion looks the best. The region is filled in very well due to the diffusion process and the texture of the sky is continued into the region. The isotropic diffusion method, again leaves a slab of nearly constant intensity behind.

The inpainting based on the averaging kernels also performs very well. This has been a trait that has been observed throughout the project. The kernels perform very well in almost every condition.

(a) Original Image.

(b) Mask to remove building

(c) Median Filtering

(d) Convolution Based on Gradient

(e) Anisotropic Diffusion

(f) Isotropic Diffusion

(g) Kernel 1

(h) Kernel 2

8 Conclusion

By observing a large amount of output images, we can now summarize the characteristics of these methods.

First, the median filtering method and diffusion filtering convolution method are the fastest ones. The gradient filtering method is also pretty fast, as we can see from the data in the previous section. However, its still a little bit slower than the previous two because it involves more calculations. Diffusion methods are the slowest. Therefore, if our inpainting system is on real-time devices that request fast processings, we may prefer the convolution-based methods here.

Second, the diffusion methods usually have smaller MSEs from original images. The diffusion filtering method also has small MSEs in most cases. The gradient filtering method does slightly worse in this aspect. The median filtering method has large MSE values in many cases. Therefore, the median filtering is only good for crude processings. If we need finer restored images, we will need to use diffusion methods.

Finally, each method has advantages and disadvantages in image smoothness and texture. The median filtering method is not very good at restoring texture because it can only roughly estimate the damaged pixel value according to the neighbor pixels. Although the gradient filtering method has larger MSE values than the median filtering method in some test cases, it generally produces better smoothness and texture in output images. This makes sense because this method does more analysis on the relations between damaged pixels and their neighbouring good pixels.

Like the gradient filtering method, diffusion methods also diffuse from good pixels to damaged pixels and thus produce natural smoothness in most cases. However, when the damaged region is too thick, it will be hard for it to completely diffuse through the damaged regions. As a result, in the test case of the Lena image covered with lots of text, we can see lots of distracting streaks. This is because the fonts are too thick to be completely restored by diffusion processes. Therefore, the diffusion methods can definitely handle lots of damaged pixels in one image. Nevertheless, the distributions of these damaged pixels shouldnt form thick regions for diffusions to work well.

The diffusion filtering method is good at keeping edges and texture, like Lenas hair. However, its filter kernels behave like averaging filters. When the intensities around the damaged regions are different, it will mix these pixel values and may create annoying shiny spots, like what happens in the test case of the Lena image with a black square on her shoulder.

9 Results for Method 1

(a) Corrupted Image. (b) Median Filtered, MSE = 0.037, PSNR = 62.37

Figure 2: Median Filtering on lena0.png

(a) Corrupted Image. (b) Median Filtered, MSE = 2.53, PSNR = 44.09

Figure 3: Median Filtering on lena1.png

(a) Corrupted Image. (b) Median Filtered, MSE = 1.61, PSNR = 46.07

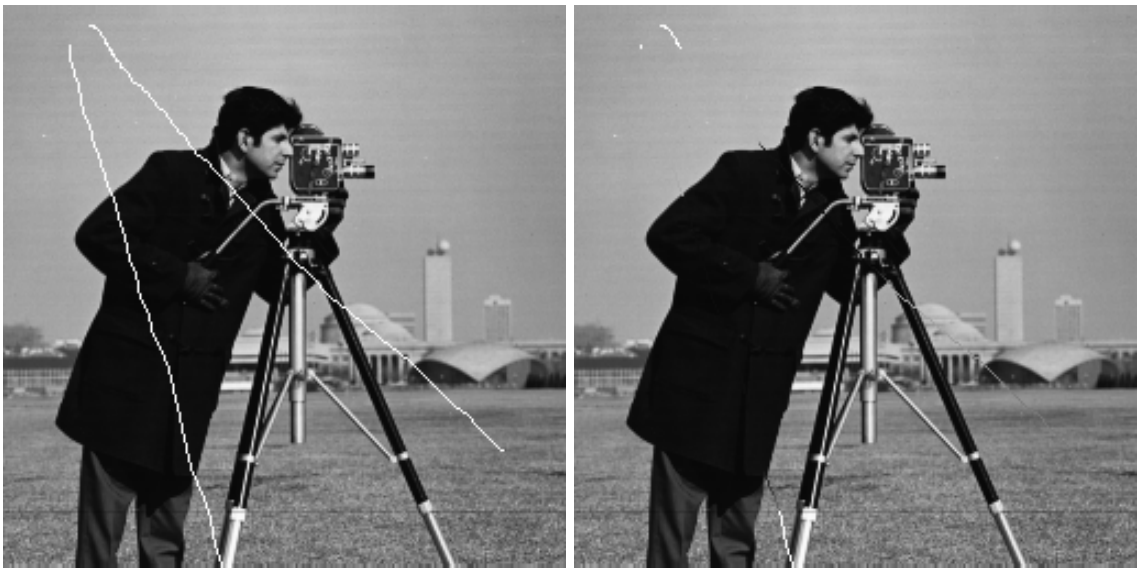
Figure 4: Median Filtering on lena2.png

(a) Corrupted Image. (b) Median Filtered, MSE = 0.213, PSNR = 54.83

Figure 5: Median Filtering on lena3.png

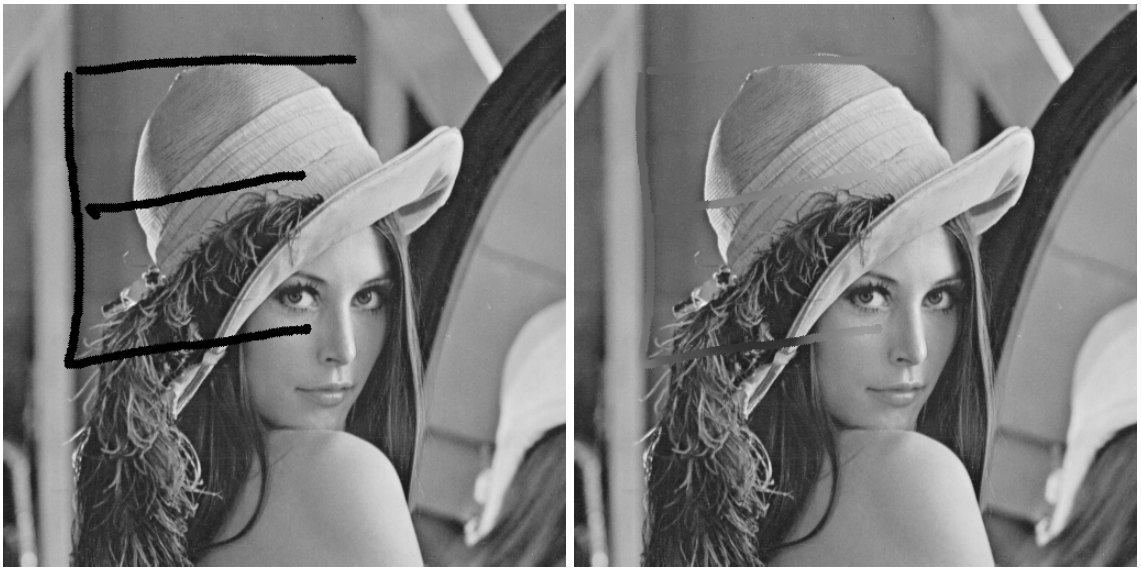
(a) Corrupted Image. (b) Median Filtered, MSE = 0.056, PSNR = 60.57

Figure 6: Median Filtering on lena4.png



(a) Corrupted Image. (b) Median Filtered, MSE = 19.92, PSNR = 35.13

Figure 7: Median Filtering on cameraman.png



(a) Corrupted Image. (b) Median Filtered, MSE = 24.18, PSNR = 34.29

Figure 8: Median Filtering on lenaece.png

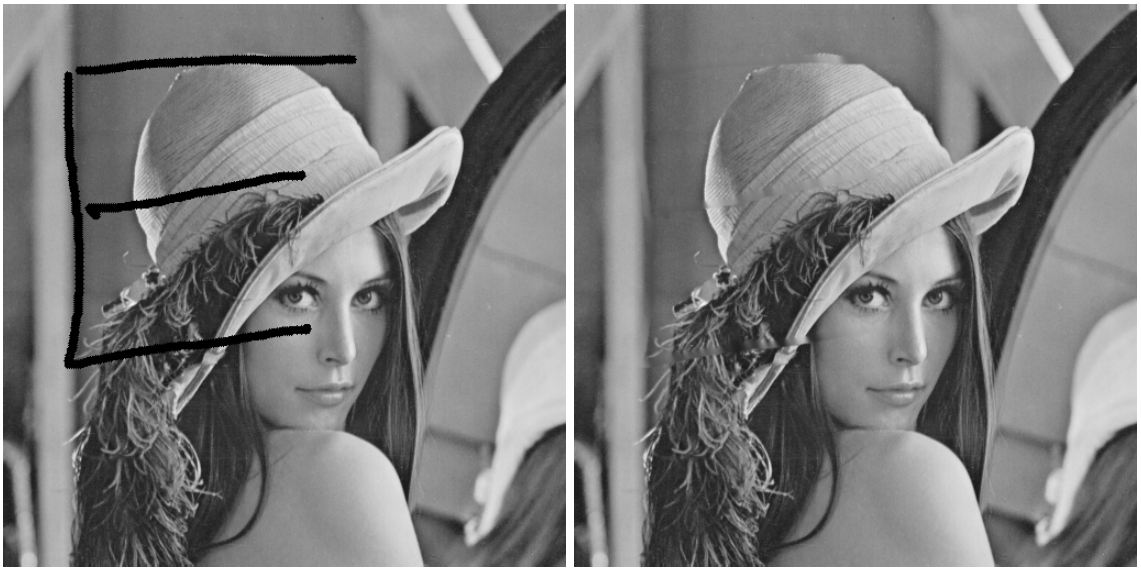
(a) Corrupted Image. (b) Median Filtered, MSE = 267.53, PSNR = 23.85

Figure 9: Median Filtering on letter.png

Table 1: Running Times for Algorithm

Image	Running Time(seconds)
lena0.png(512x512	0.09
lena1.png(512x512)	0.1
lena2.png(512x512)	0.1
lena3.png(512x512)	0.09
lena4.png(512x512)	0.1
cameraman.png(256x256)	0.1
lena_ece.png(512x512)	1.79
letter.png(512x512)	14.78

10 Results for Method 2



(a) Corrupted Image. (b) Gradient Filtered, MSE = 9.31, PSNR = 38.44

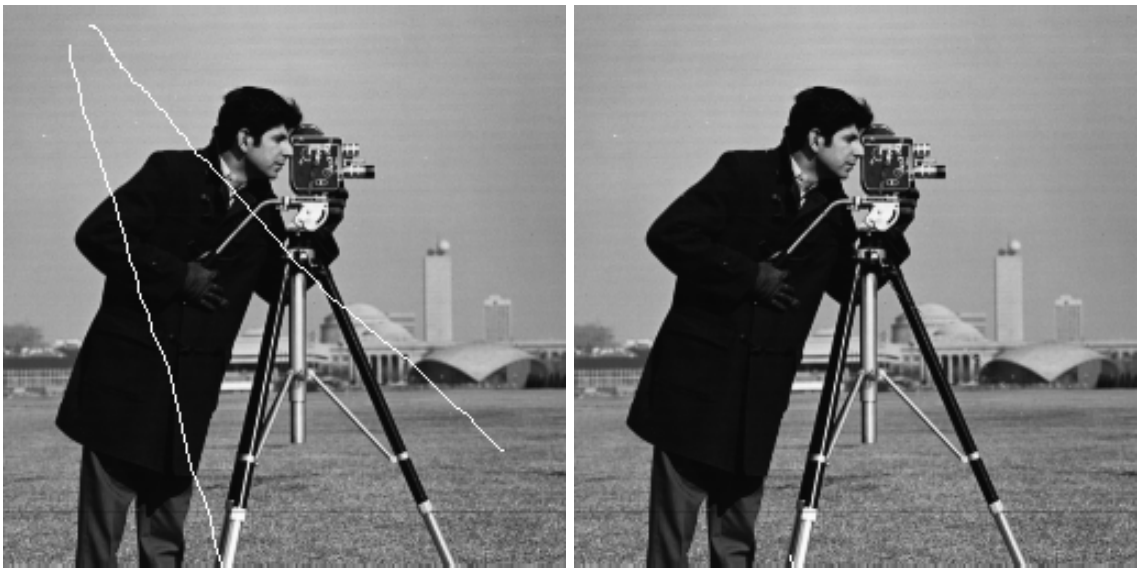
Figure 10: Gradient Filtering on lenaece.png

(a) Corrupted Image. (b) Gradient Filtered, MSE = 0.19, PSNR = 55.28

Figure 11: Gradient Filtering on lena3.png

(a) Corrupted Image. (b) Gradient Filtered, MSE = 3.58, PSNR = 42.58

Figure 12: Gradient Filtering on lenablack.png



(a) Corrupted Image. (b) Gradient Filtered, MSE = 4.51, PSNR = 41.58

Figure 13: Gradient Filtering on cameraman.png

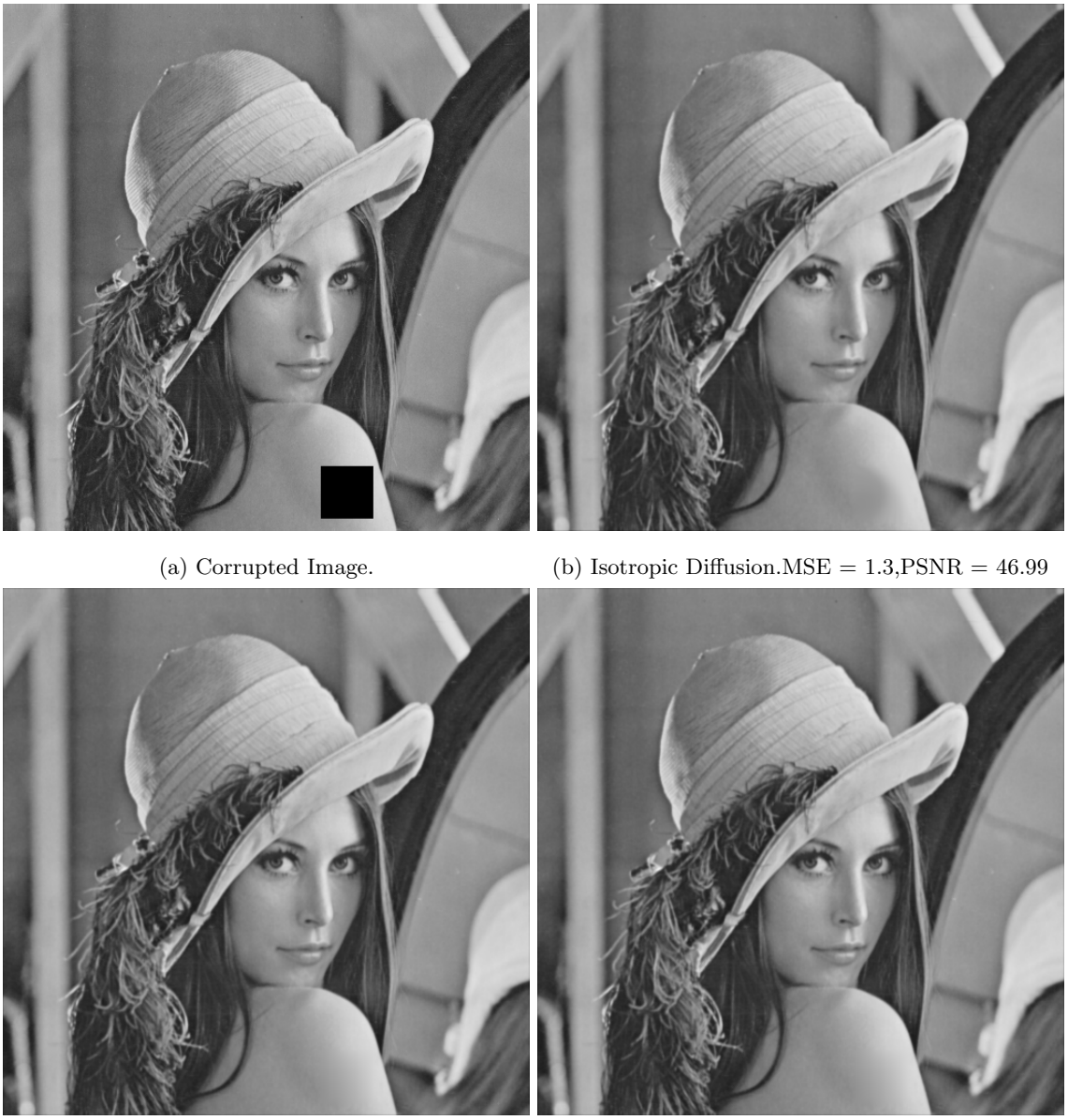
(a) Corrupted Image. (b) Gradient Filtered, MSE = 69.93, PSNR = 29.68

Figure 14: Gradient Filtering on letter.png

Table 2: Running Times of the Algorithm

Image	Running Time(seconds)
lena_ece.png(512x512)	0.63
lena3.png(512x512)	0.01
lenablack.png(512x512)	0.1
camerman.png(256x256)	0.22
letter.png(512x512)	3.37

11 Results for Method 3



(a) Corrupted Image.

(b) Isotropic Diffusion.MSE = 1.3,PSNR = 46.99

(c) Anisotropic Diffusion K=2. MSE = 0.17,PSNR = = 55.83

(d) Anisotropic Diffusion K=10. MSE = 0.1699,PSNR = 55.83

Figure 15: PDE inpainting for lenapatch1.png.

(a) Corrupted Image.

(b) Isotropic Diffusion. MSE = 6.35,PSNR = 40.13

(c) Anisotropic Diffusion. MSE = 6.35,PSNR = 40.13

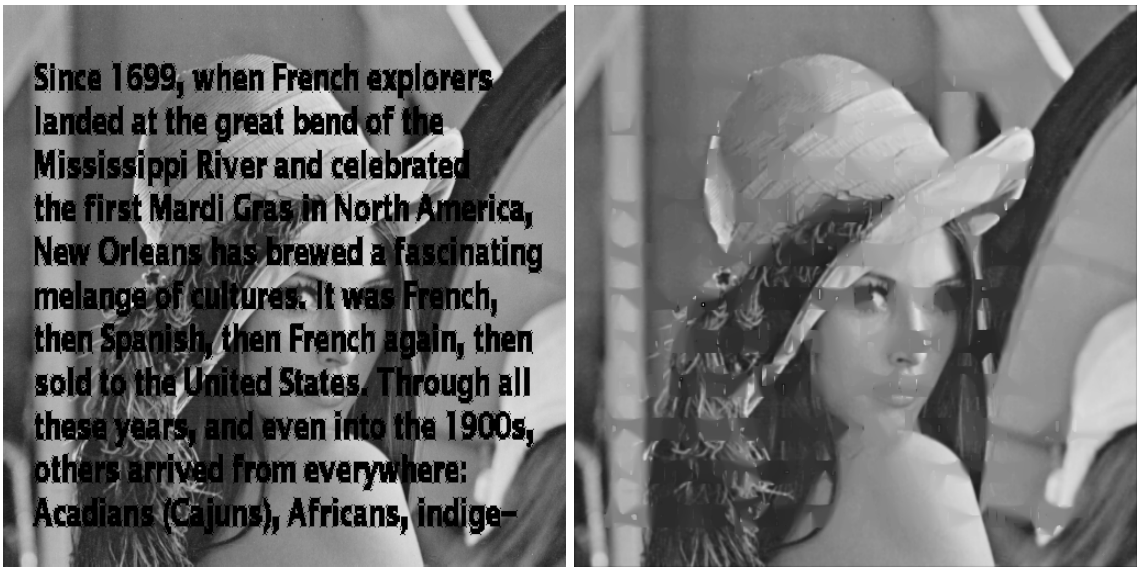
Figure 16: PDE inpainting for cameraman.png

(a) Corrupted Image.

(b) Isotropic Diffusion. MSE = 27.72,PSNR = 33.70

(c) Anisotropic Diffusion. MSE = 8.34,PSNR = 38.91

Figure 17: PDE inpainting for lenaece.png



(a) Corrupted Image.

(b) Isotropic Diffusion. MSE = 172.15,PSNR = 25.77

(c) Anisotropic Diffusion. MSE = 93.38,PSNR = 28.42

Figure 18: PDE inpainting for lenawords.png

12 Results for Method 4

(a) Corrupted Image.

(b) Using Kernel 1. MSE = 18.28,PSNR = 35.51

(c) Using Kernel 2. MSE = 18.26,PSNR = 35.51

Figure 19: Inpainting lenapatch.png using diffusion filters.

(a) Corrupted Image.

(b) Kernel 1. MSE = 1.42,PSNR = 46.61

(c) Kernel 2. MSE = 1.53,PSNR = 46.28

Figure 20: Inpainting cameraman.png using diffusion filters.

(a) Corrupted Image.

(b) Kernel 1. MSE = 5.64,PSNR = 40.61

(c) Kernel 2. MSE = 5.64,PSNR = 40.61

Figure 21: Inpainting lenaece.png using diffusion filters.

(a) Corrupted Image. (b) Kernel 1. MSE = 30.07,PSNR = 33.35

(c) Kernel 2. MSE = 29.63,PSNR = 33.41

Figure 22: Inpainting lenawords.png using diffusion filters.

13 Appendix

We have 4 packages, each of which contains codes, source papers and test images for each of the method. Here are some clarifications and compiling instructions for these submitted materials.

13.1 Median - Method 1

The pair of all input test images and corresponding output images can be easily identified by file names. The code we use to implement the median filtering method is in main.cc.

To compile the program, simply type make. To run the program, type **./main input1 input2 input3 input4 input5 input6**. The first input is the file name of the input image. The second one is the file name of the output image. The third and fourth inputs are the horizontal and vertical coordinates of the top left corner of the searching area. The fifth and sixth inputs are the horizontal and vertical coordinates of the bottom right corner of the searching area.

For example, in the test case of lena3, we would type **./main lena3.png lena_ output3.png 140 70 160 90**. To clear out current program objects, type make clean. The file testimage.txt, contains the coordinates of all feature points for the searching areas in all test cases. The PDF file is the paper we study this method, which is referenced in the Work Citations part.

13.2 Convolution gradient - Method 2

Everything is very similar to the Median folder. To run the program, type **./main input1 input2 input3 input4 input5**. The first input is the file name of the input image. The second one is the file name of the output image. The third and fourth inputs are the horizontal and vertical coordinates of the start searching point. The last element defines the searching range in both the horizontal and vertical directions.

For example, in the test case of letter.png, we would type **./main letter.png letter_ output.png 2 2 500**. The file testimage.txt, contains the coordinates of the start searching points and the searching ranges for all test cases.

13.3 PDE and Diffusion Filter - Method 3, 4

The scripts to run these are in the folders listed in the title. To run the diffusion code open inapainting_ bertalmi, and create any of the masks and the images. Then in the last 2 cells, you can choose to run diffusion or the convolution based method.

To run the PDE based method with diffusion call inpaint(I, mask, dt, num_ iter) where I is the image to inpaint, dt is the time step. You can also call anisotropicDiffusion(I, dt, k, mask, option). If option is 1 it will run isotropic diffusion with time step dt and if option is 2 it will run anisotropic diffusion with time step dt and parameter k.

14 Works Cited

1. Awati, Anupama Sanjay, and Meenakshi Patil. "Digital Image Inpainting Based on Median Diffusion and Directional Median Filtering." International Journal of Computer Applications, Mar 2015.

2. Noori, H., et al. "A Convolution Based Image Inpainting." 1st International Conference on Communications Engineering, 2010.

3. M. Bertalmio, G. Sapiro, V. Caselles, C. Ballester, "Image inpainting", Proceeding of SIGGRAPH. 2000 computer graphics processing, pp. 417-424.

4. M. Oliveira, B. Bowen, R. Mckenna, Y.S. Chang, "Fast digital image inpainting", proc. VIIP2001, pp. 261-266, 2001

5. P. Perona and J. Malik Scale-space and edge detection using anisotropic diffusion. IEEE-PAMI 12, pp. 629-639, 1990.