

---

# FINE-GRAINED DESIGN AND IMPLEMENTATION

Based on the coarse-grained design of the system proposed for restoration of image defects presented in the preceding chapter, this chapter provides a fine-grained design of the system as well as the implementation details. The problems encountered while implementing the proposed design and the solutions used to overcome these problems are also described.

## 4.1 Implementation Platform

The system interface has been developed using the Visual C++ development environment running under the Windows operating system. Since the system uses an object-oriented approach, Visual C++ is very well suited for this purpose since the language provides the basic object-oriented facilities i.e. the ability to declare generic classes, to use inheritance, aggregation and other concepts. Furthermore, the Visual C++ development environment provides a set of tools that enables developers to create Windows applications with ease and speed. With MFC (Microsoft Foundation Classes), programming is even made easier since the use of libraries enables code reuse and a variety of class libraries are available to support windows, buttons and dialog controls. The basic coding is already done such that developers can concentrate more on specific programming tasks. Moreover, MFC provides several facilities to manipulate and display images. Visual C++ has also an extensive debugging environment and a range of debugging tools that help with program development.

## 4.2 Class Description

A **class** is the descriptor for a set of objects with similar structure, behavior, and relationships. Class diagrams show the static structure of the model, in particular, the things that exist (such as classes and types), their internal structure, and their relationships to other things. Figure 4.1 shows the class diagram for the system proposed.

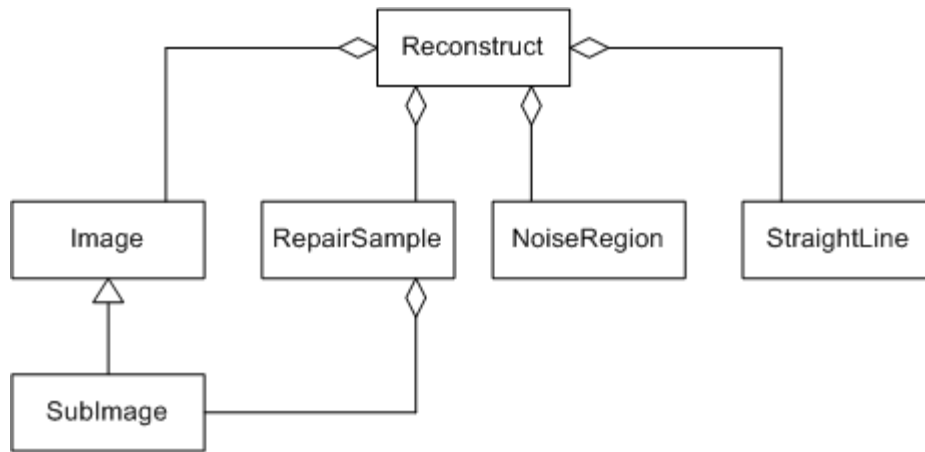


Figure 4.1: Class diagram

### 4.2.1 Reconstruct Class

This is the main class which contains methods to restore the input image.

#### Attributes

	Description
Image imgOriginal	Original image that the user inputs to the system
NoiseRegion noise	Stores noise region details
RepairSample subimgs	Stores repair and sample subimages details
CStraightLine straightline	Stores line details
int WidthEstimate	Width of region used for estimation for water blotch removal
int ShiftDistance	Distance estimate region is shifted from the border in the water blotch removal
int LengthArray	Length of array used for restoring water blotch contour

## Methods

**(i) public void Basic (int iteration)**

Applies the basic method of the image noise algorithm on the original image. The function takes as input the number of iterations to be applied.

**(ii) public void BlotchRemove ()**

Applies the semi-transparent blotch removal method on the original image.

**(iii) public int noiseErase(CPoint erasept[], int x, int y, int brushsize)**

Fills the empty array *erasept* [] passed as input with coordinates of pixels found in the square region centered on pixel  $(x, y)$  of length *brushsize*. Returns the number of noisy pixels erased.

**(iv) public void noisepaint(int x, int y, int brushsize)**

Adds all pixels found in the square region, centered on pixel  $(x, y)$  of length *brushsize*, to the noise array.

**(v) public void LineCorrection ()**

Applies the straight line restoration method on the original image.

**(vi) public void SplitFrequency (int iteration)**

Applies the split frequency method of the image noise algorithm on the original image. The function takes as input the number of iterations to be applied.

**(vii) public void SoftScratch (int iteration)**

Applies the soft scratch method of the image noise algorithm on the original image. The function takes as input the number of iterations to be applied.

**(viii) public void segment (int x, int y, double R, double G, double B)**

Finds all pixels that have close intensity values and which are connected to the initial pixel  $(x, y)$  of intensity value  $(R, G, B)$  and marks them as noise pixels. If the image is grayscale, only intensity value  $R$  is considered.

(ix) **protected double getVariance (double \*list, double mean, int number)**

Returns the variance of the numbers in *list*. *mean* is the average of the numbers. The last parameter is the number of numbers in *list*.

(x) **protected double getMean (double \*list, int number)**

Returns the mean of the numbers in *list*. The last parameter is the number of numbers in *list*.

### 4.2.2 Image Class

This class provides the necessary attributes to store image details as well as the methods to perform appropriate operations on an image.

#### Attributes

	Description
int width, height	Width and height of the image in pixels
int maxgray	Maximum gray level
int bands	Number of bands (bands = 1 for grayscale images; bands = 3 for color images)
comp **matrix[3]	Three-dimensional array of type complex to store intensity values (real and imaginary) for each band

#### Methods

(i) **public void clip (int bnd)**

Clips all intensity values in the color band *bnd* to values between 0 to the maximum gray level.

(ii) **public void ForwardFFT (int bnd)**

Performs a forward FFT on the specified band of the image: pre-processes the data in *matrix* and calls function *FFT*. The transformed values are stored in the 2-D complex array *matrix*.

(iii) **protected void FFT (double data[], unsigned long nn[], int ndim, int isign)**

The first parameter *data[]* is the data array of size  $(2*nn)$  starting at index 1. *nn* is number of complex data points in the array in each dimension. *ndim* is the number of dimensions of the data array; in this case, since we are dealing with only images, this parameter is always 2. *isign* takes values  $\pm 1$  (1 when forward FFT needs to be performed, -1 when inverse FFT needs to be performed).

(iv) **public void InverseFFT (int bnd)**

Performs an inverse FFT on the specified band of the image: calls function *FFT* and applies appropriate post-processing on the resulting data. The final values are stored in the 2-D complex array *matrix*.

(v) **protected int powerOf2 (int x)**

Returns the value of *n* where  $x = 2^n$  and  $n \geq 0$ . If *x* is not a power of 2, then the function returns the value of *n* that rounds *x* to the next power of 2.

(vi) **public void ReadImg (ifstream ifs)**

Reads in and stores values in image object from an image file. *ifs* is the handle to the image file.

(vii) **protected void Swap (double &a, double &b)**

Swaps values of *a* and *b*.

(viii) **public void WriteImg (ofstream ofs)**

Writes values of image object to an image file. *ofs* is the handle to the image file.

### 4.2.3 SubImage Class

This class is derived from the class *Image* and it is used to capture repair and sample subimages attributes for the image noise removal algorithm. Since the class is a child of the class *Image*, it inherits the attributes and methods of the parent class. Additional attributes specific to the class *SubImage* are described below.

**Attributes**

	Description
int left, top, right, bottom	The leftmost, topmost, rightmost, bottommost position of the subimage within the image
double ** spectrum	The Fourier spectrum (or magnitude) values obtained after applying FFT on the subimage
SubImage *Next	Used in the linked lists: it points to the next subimage in the list or to NULL if the object is found at the end of the list.

**4.2.4 NoiseRegion Class**

This class stores details about the noise region(s) that the user selects and provides the appropriate operations to manipulate them.

**Attributes**

	Description
CPoint *Noisepix	The coordinates of all noise pixels
int NoisepixCount	Number of noise pixels
CPoint top, bottom, left, right	Coordinates of topmost, bottommost, leftmost and rightmost pixel in noise region
CPoint boundarypix	The coordinates of the boundary pixels in the noise region
int numboundarypix	Number of pixels on boundary
bool ** NoiseBitmap	Keeps track of noise pixels location
int height, width	Height and width of noise bitmap

**Methods**

(i) **public void AddNoise (CPoint pt)**

Adds the pixel *pt* to the array *Noisepix*.

**(ii) public int bordergradient(int img[])**

Returns the gradient direction of the contour of the noise region at one point where *img[]* is the 3x3 neighborhood of the noise pixel under consideration.

**(iii) public void BuildNoiseBitmap ()**

Scans the noise array and sets values of the noise bitmap to *true* if the pixel is a noise pixel, otherwise values are set to *false*. The function also performs an opening operation on the newly built noise bitmap.

**(iv) protected void checkConnectivity(int i, int j, int \*\*labelimg, int ID)**

Labels all pixels in *labelimg* with the value of *ID*. The pixels labeled are those whose corresponding pixels in the noise bitmap are connected to the pixel (*i, j*).

**(v) public int countRegions()**

Returns the number of noise regions selected by the user. The function supplies the appropriate parameter to the function *label*.

**(vi) protected void defineBoundary ()**

Finds topmost, bottommost, leftmost, rightmost pixel in the noise region.

**(vii) public void deleteNoise (CPoint list[], int num)**

Removes all pixels found in the array *list* from the *Noisepix* array. *num* is the number of noisy pixels to delete.

**(viii) protected void Dilation()**

Applies a dilation operation to the noise bitmap.

**(ix) protected void Erosion()**

Applies an erosion operation to the noise bitmap.

**(x) protected void identifyBoundary (int topx, int topy, bool \*\*region)**

Identifies the pixels found on the boundary of the noise region and stores the corresponding pixel coordinates in the array *boundarypix*. (*topx, topy*) is the starting pixel used in the border tracing algorithm and *region* is the noise bitmap.

**(xi) protected int label(int \*\*labelimg)**

Scans the noise bitmap and assigns an ID to each noise region found in the bitmap. Returns the number of noise regions encountered.

**(xii) public bool traceBorder()**

Builds the noise bitmap, defines the noise boundaries and identifies pixels. Calls protected functions *BuildNoiseBitmap*, *defineBoundary* and *identifyBoundary*.

**(xiii) protected void updateNoisepix (int index)**

Shifts all pixels found in the noise array one position to the left, starting from position *index*. The function is called when a pixel is deleted from the noise array.

### 4.2.5 RepairSample Class

This class manages the two linked lists that stores repair and sample subimages information.

#### Attributes

	Description
SubImage * RootRepair	Points to first object in repair subimage linked list
SubImage * RootSample	Points to first object in sample subimage linked list
int numRepair	Number of repair subimages in linked list
int numSample	Number of sample subimages in linked list

#### Methods

**(i) public SubImage \* GetSamplePtr()**

Returns a pointer to the last sample subimage object in the linked list.

**(ii) public void AddSample(SubImage \*img)**

Adds a new sample subimage object *img* to the sample subimage linked list.

**(iii) public void AddRepair (SubImage \* img)**

Adds a new repair subimage object *img* to the repair subimage linked list.

### 4.2.6 StraightLine Class

This class stores line details and provides methods to perform basic operations on a line.

#### Attributes

	Description
CPoint linestart, lineend	Coordinates of start and end points of the line
CPoint * line	All pixels found on the line
int lineLength	Number of pixels on the line
int lineWidth	Width of the line
double gradient	Gradient of the line

#### Methods

(i) **void ComputeGradient()**

Computes the gradient of the line and stores the result in the attribute *gradient*.

(ii) **void buildLine()**

Computes the positions of the pixels found on a line using the line drawing algorithm DDA, given the starting and ending pixel locations of the line. The pixel coordinates are stored in the array *line*.

## 4.3 Opening an Image

To read in any type of image – grayscale or color, the same function *Image::ReadImg* is used. The function checks the identifier on the first line of the image file to determine whether the image is a grayscale one (P5) or a color one (P6). If the image being opened is not a PGM or a PPM image, the open operation exits. If the image is a grayscale one, only one band is created. If the image is a color one, three bands are created: the first one is for the red band, the second is for the green band and the third is for the blue band. The value of the variable *band* in the image class is also set accordingly.

The flowchart below shows the steps involved in reading a PGM or PPM image.

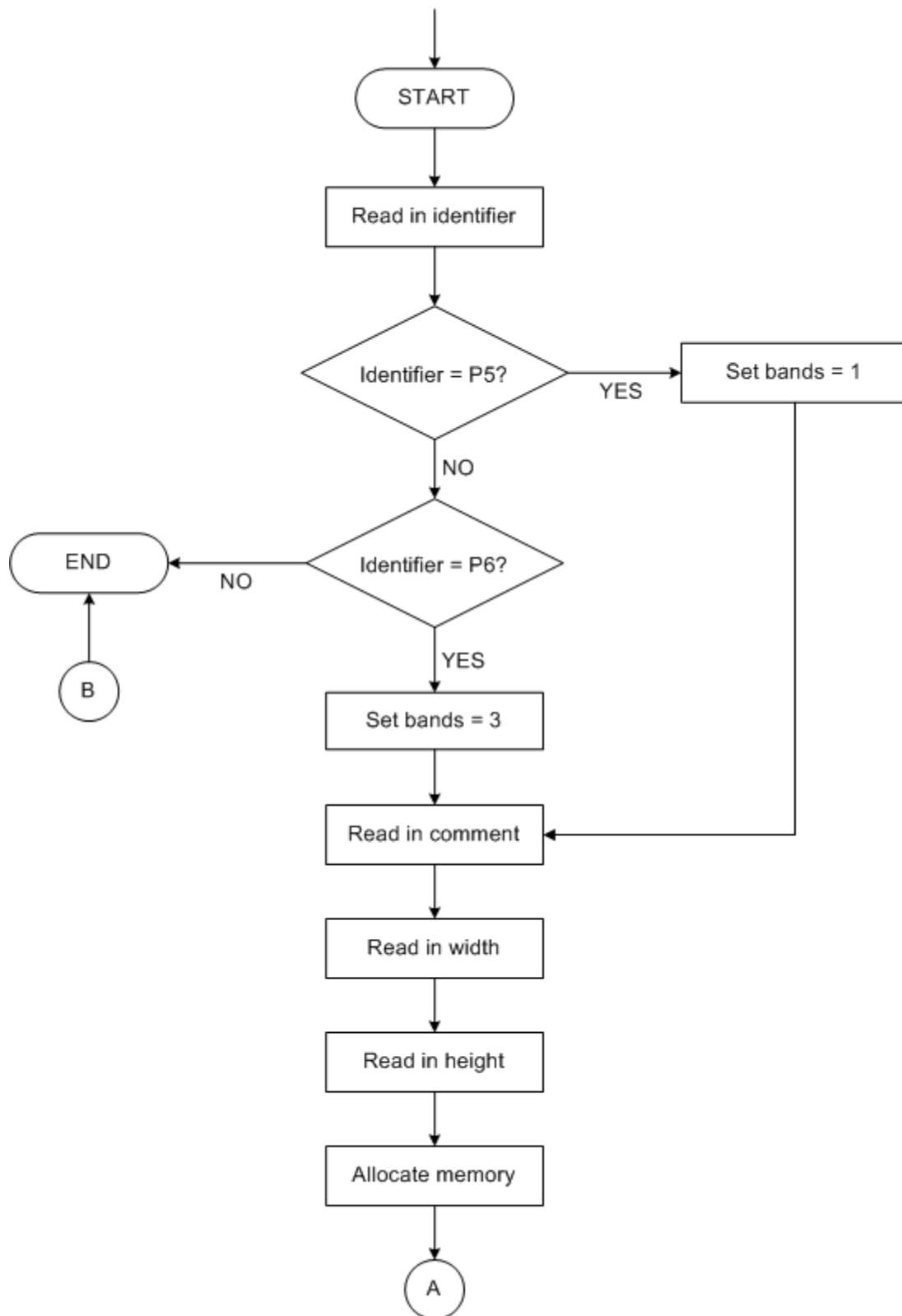


Figure 4.2: Opening an image

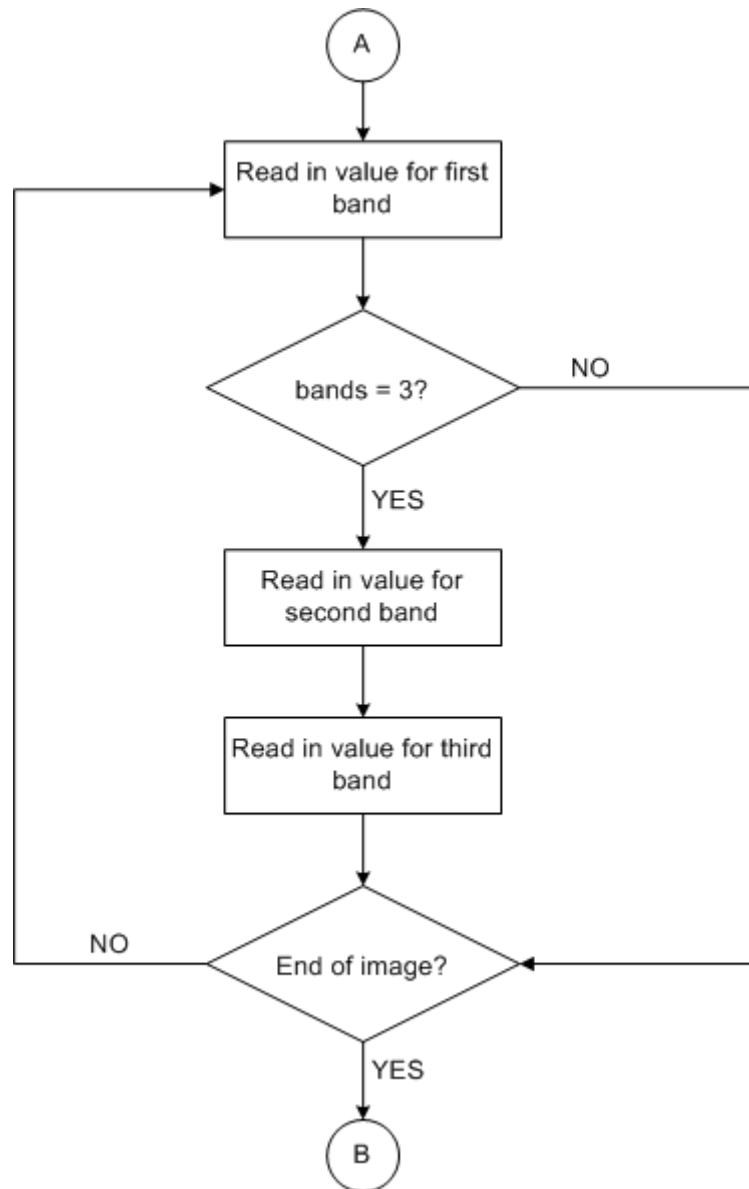


Figure 4.2 (continued)

#### 4.4 Automatic Noise Selection

The algorithm for automatic noise selection starts with an initial pixel, of intensity  $I$ , that the user has selected in the noise region. Consequently, all pixels which are directly or indirectly connected to this pixel and which have an intensity close to  $I$  are labeled as noisy pixels too. 4-connectivity is used in this process. Hence, the algorithm checks each of the 4-neighbors of the initial pixel to see if they are part of

the noise region too. The algorithm recursively performs this operation, each time one of the 4-neighbors of the initial pixel becoming itself the initial pixel.

The algorithm stops if any of these three conditions are met:

- (i) The initial pixel is beyond the image borders.
- (ii) The intensity difference between the initial pixel intensity and  $I$  is greater than a certain threshold.
- (iii) The initial pixel has already been labeled as noisy.

Figure 4.3 on the next page illustrates how the automatic noise selection method works.

#### **4.5 Manual Noise Selection**

For the paintbrush tool, when the user selects a pixel in the image, the status of the surrounding pixels is set to noisy, if it was not already the case. The size of the surrounding region depends on the brush size as selected by the user. For instance, if a brush size of 5 pixels has been chosen, then all pixels inside the rectangular region of size 5x5, centered on the pixel selected, are added to the list of noise pixels. The noise eraser tool works in the same way except that the surrounding pixels are removed from the noise list if they have already been identified as noise pixels.

Figure 4.4 on page 72 illustrates how the manual noise selection method works.

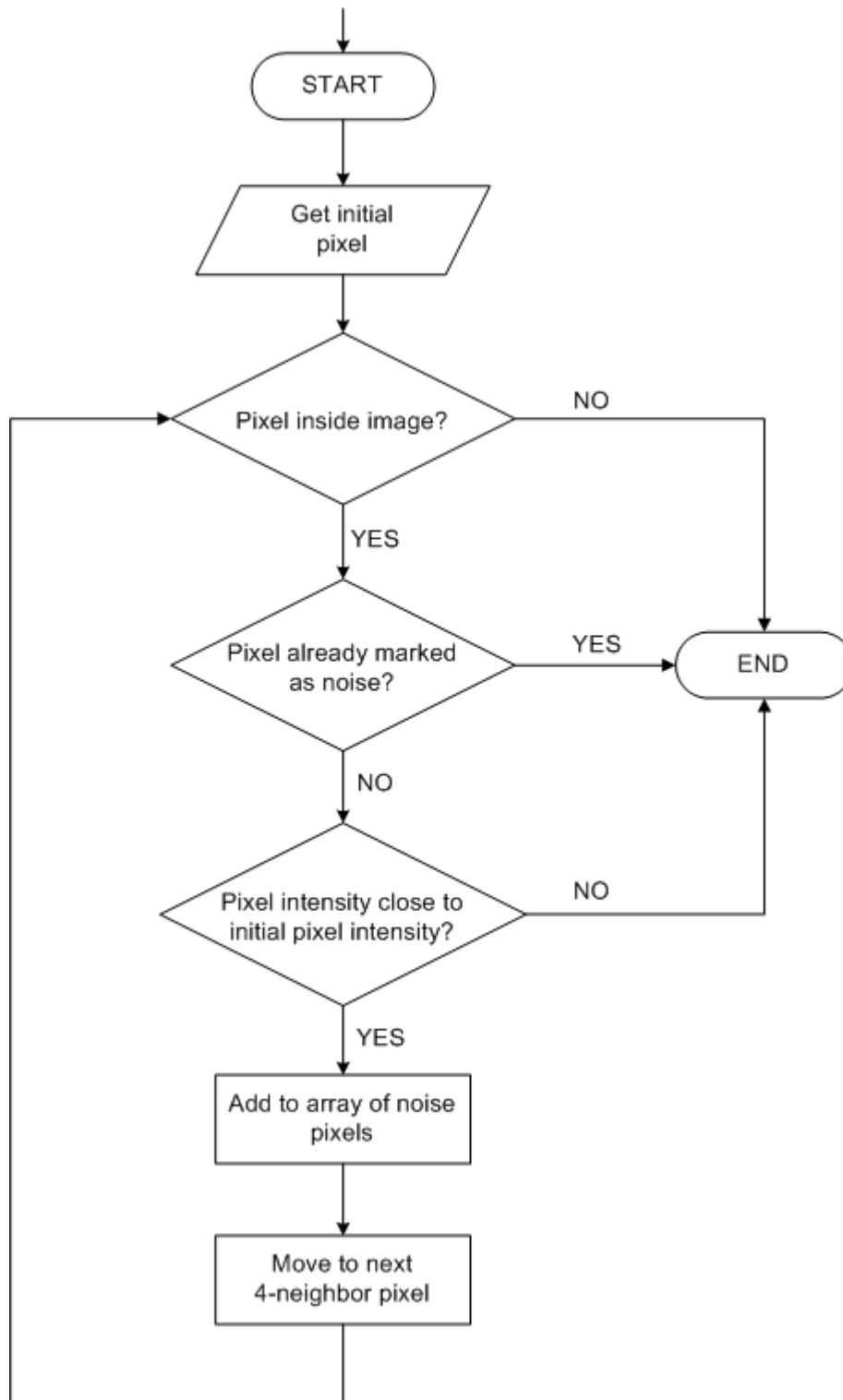


Figure 4.3: Automatic noise selection

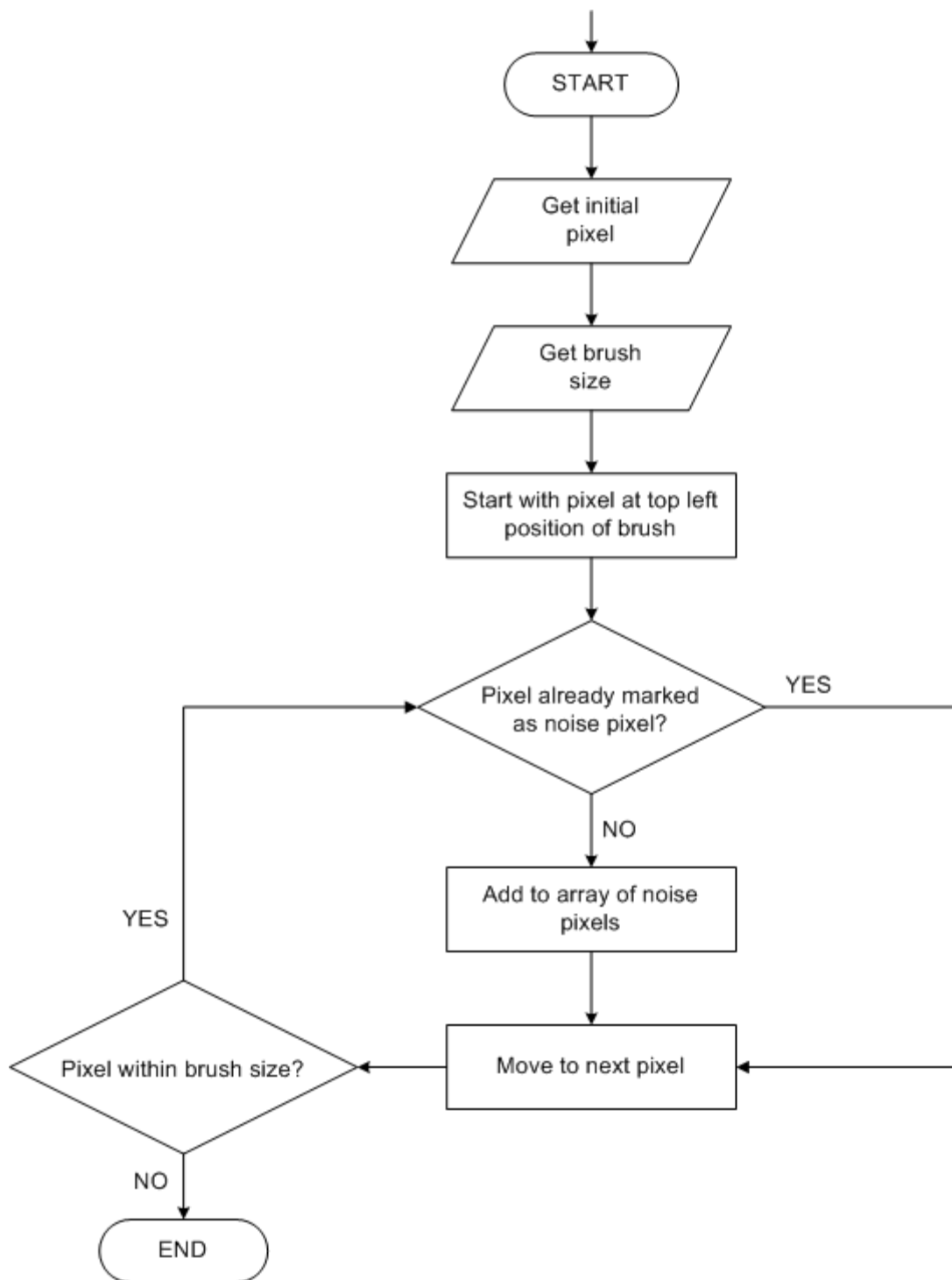


Figure 4.4: Manual noise selection

## 4.6 Semi-Transparent Blotch Restoration

### 4.6.1 Step 1

#### Identifying pixels for estimation of uncorrupted image

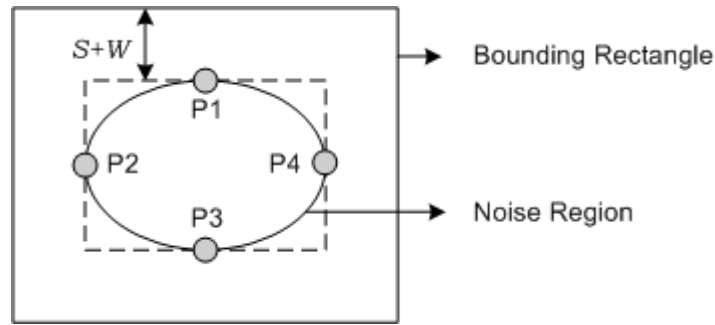
In the first step of the method, the pixels to be used in the estimation of the uncorrupted region need to be identified. These pixels should lie at a distance of at least  $S$  but less than  $S+W$  from the border of the semi-transparent blotch, where  $S$  is the shift distance and  $W$  is the width of the region to be used for estimation. Therefore, if the minimum distance between each pixel found outside the noise region and the noise region is calculated, only pixels whose minimum distance is between  $S$  and  $S+W$  are used for estimation.

The distance  $d$  between two pixels of coordinates  $(x_1, y_1)$  and  $(x_2, y_2)$  is given as:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

To find the minimum distance between a pixel (outside the noise region) and the noise region, the distance between that pixel and *each* pixel in the noise region can repeatedly be computed. However, if the number of noisy pixels is high, then this method of finding the minimum distance can take a lot of time. To optimize coding, the minimum distance between a pixel outside the noise region and each pixel on the *inner border* of the noise region is calculated instead. This significantly reduces the number of computations since the number of pixels making up the boundary is usually much smaller than the total number of pixels making up the noise region, especially when the noise region is large.

To further reduce processing, the topmost, leftmost, rightmost and bottommost pixel of the noise area can be used to define a bounding rectangular region that encloses all the noisy pixels. The rectangular region is extended by a distance of  $S+W$  pixels, as shown in Figure 4.5, such that only pixels within this extended region are used for minimum distance calculation.



**Figure 4.5:** Bounding rectangular region.  $P1$ ,  $P2$ ,  $P3$  and  $P4$  represent respectively the topmost, leftmost, bottommost and rightmost pixels of the noise region.

Initially, the array that contains all pixel positions forming part of the noise region was used to determine whether a pixel is noisy or not for the above steps. However, this process required a complete loop through the whole array in cases where the pixel was non-noisy. To decrease computation time, especially when the noise region is large, a **noise bitmap** has been built. This noise bitmap is of the same size as the original image, but at each pixel location we have only two values: *true* if the pixel is part of the noise region, otherwise the value is *false*. Hence, to determine if a pixel is noisy or not, we simply need to check whether its corresponding value in the noise bitmap is set to *true*, considerably reducing computation time.

The steps to find the minimum distance from the border are as follows:

1. Declare a temporary variable  $min\_dist$  to store coordinates of a pixel
2. For each pixel  $(x, y)$  in the image, do:
  - For each pixel  $(xb, yb)$  on the inner boundary of the semi-transparent blotch, do:
    - If this is the first pixel being encountered,  $min\_dist = \text{first pixel}$
    - Else, if distance between  $(x, y)$  and  $min\_dist >$  distance between  $(x, y)$  and  $(xb, yb)$ ,  $min\_dist = (xb, yb)$
  - The minimum distance between  $(x, y)$  and the border =  $min\_dist$

### Calculation of parameters for additive-multiplicative model

Now that the list of pixels to be used for estimation of the uncorrupted region has been obtained, the mean and variance values for this set of pixels need to be calculated. For the corrupted region, only the pixels inside the semi-transparent blotch are used for the calculation.

The mean and variance values for a set of pixels are calculated as follows:

$$\begin{aligned} \text{Mean, } \bar{x} &= \frac{\sum x}{n} \\ \text{Variance} &= \frac{\sum (x - \bar{x})^2}{n} \end{aligned}$$

where

$n$  is the number of pixels taken into consideration

$x$  is the intensity value of each pixel in the set

### 4.6.2 Boundary Tracing

To determine the coordinates of the pixels that lie on the inner boundary of the noise region, the boundary tracing algorithm described in [SONK98] has been used. This algorithm requires the coordinates of an initial pixel located on the border to be able to proceed. This pixel has been taken to be the topmost pixel in the noise region since this pixel will automatically be on the border.

The steps for the algorithm are as follows:

1. The starting pixel (P0) is initially the topmost pixel, and a variable *direction*, whose initial value is 7, is declared. This variable stores the direction of the previous move along the border from the previous border point to the current border point. The direction notations for 8-connectivity used in the algorithm are defined in Figure 4.6.

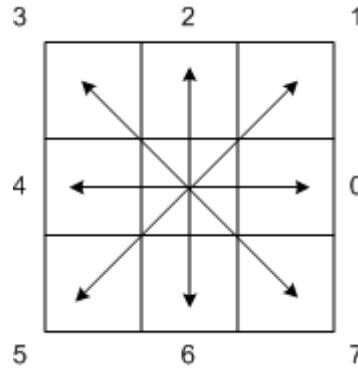


Figure 4.6: Direction notation in boundary tracing

2. The 3x3 neighborhood of the pixel P0 is searched to locate the first neighbor (P1) of P0 where P1 is also a noisy pixel. The search is done in an anti-clockwise direction, where the beginning search direction is:
  - (i)  $(direction + 7) \bmod 8$  if  $direction$  is even (rotate direction by 45° clockwise)
  - (ii)  $(direction + 6) \bmod 8$  if  $direction$  is odd (rotate direction by 90° clockwise)

Pixel P1 is then stored as a border pixel.
3. The direction of P1 with respect to P0 is the new value of  $direction$ . Step 2 is repeated with P1 as the new starting pixel.
4. The searching process is stopped when the starting pixel coincides again with its initial value (i.e. the topmost pixel).

The flowchart in Figure 4.7 illustrates the above algorithm.

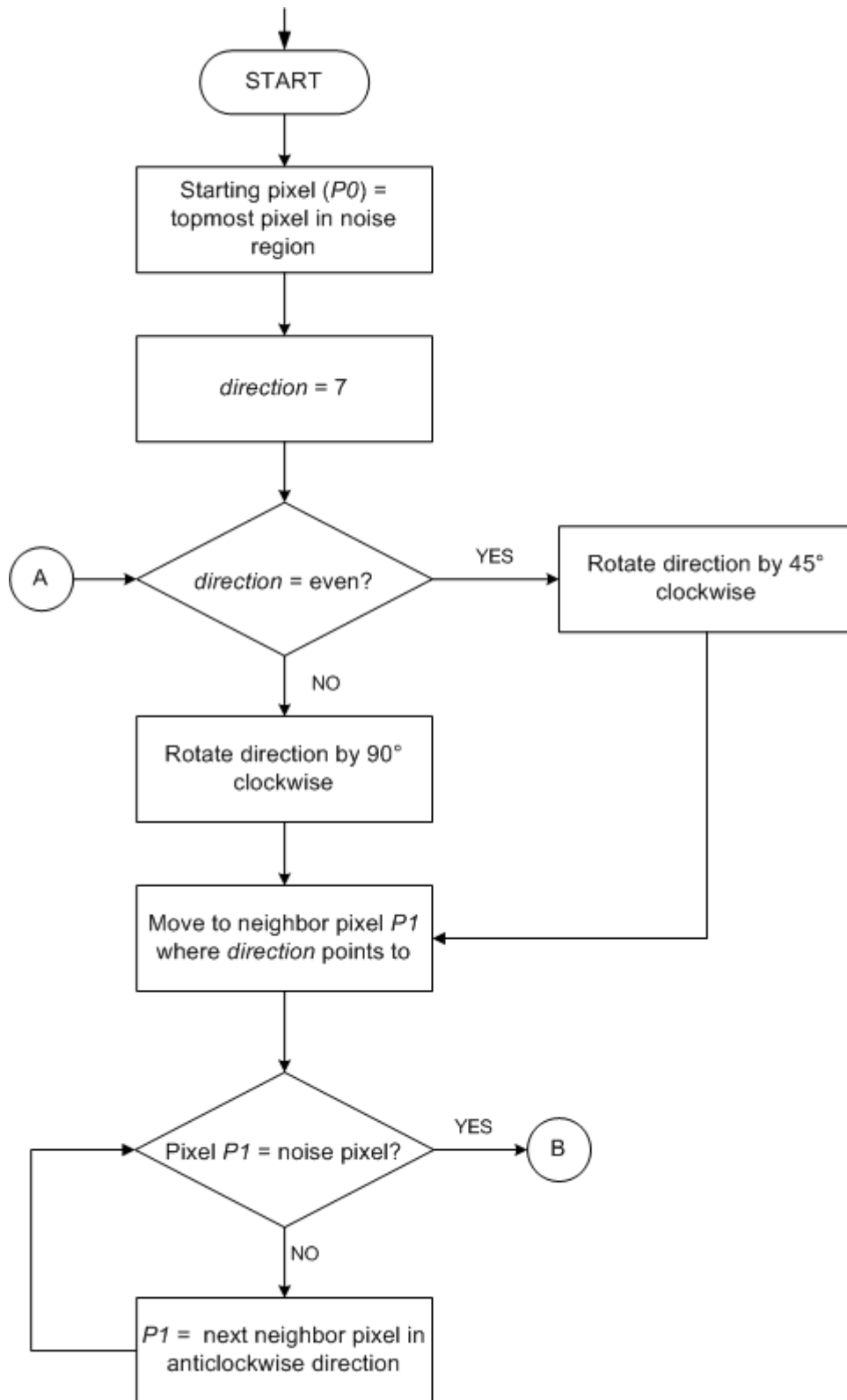


Figure 4.7: Boundary tracing algorithm

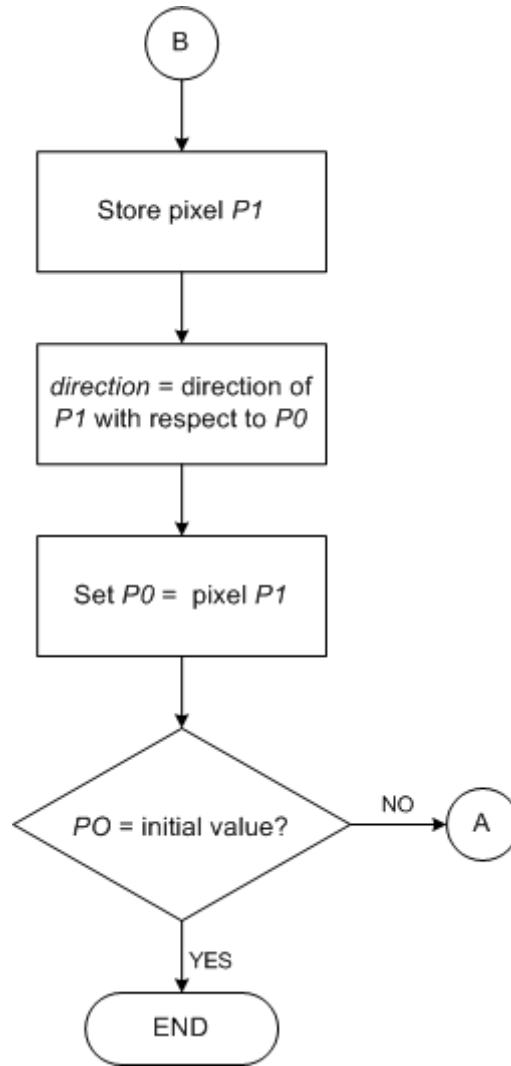


Figure 4.7 (continued)

However, the boundary tracing algorithm described above works only for regions larger than one pixel. Otherwise, the algorithm will loop forever. To overcome this limitation, a morphological opening procedure is used to eliminate any part of unit width that lies on the border of the noise region.

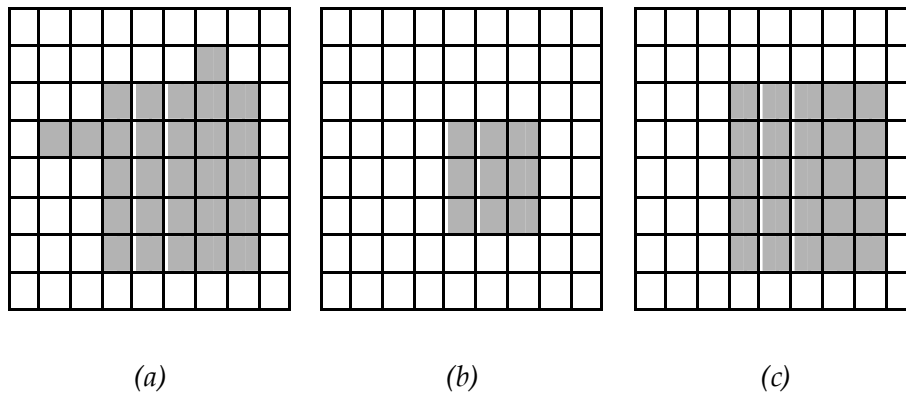
### 4.6.3 Morphological Opening

The opening operation is defined as erosion followed by dilation. **Erosion** is the term given to the process of replacing the value of a pixel in an image with the minimum value in a neighborhood, while **dilation** is the opposite process: each pixel is assigned the maximum value in the neighborhood. The erosion operation erodes

objects all round the object equally if it is applied on a binary image. Similarly, dilation operation enlarges the objects all round the object equally.

In our case, the open operation is applied to the noise bitmap (which can be considered as a binary image) and the neighborhood is taken to be the pixels inside a 3x3 structuring element. This structuring element is applied to each pixel in the image and is centered over the pixel under consideration. The erosion operation searches all pixels in this neighborhood, including the pixel under consideration, for a non-noisy pixel i.e. when a noise bitmap point has value *false*. If at least one such pixel is found, the value of the pixel at the centre of the square structure is set to *false*. Otherwise, it is left to *true*. The dilation procedure is exactly the same except that we search points that have value *true* in the neighborhood and if at least one such point is found, the replacing value is *true*.

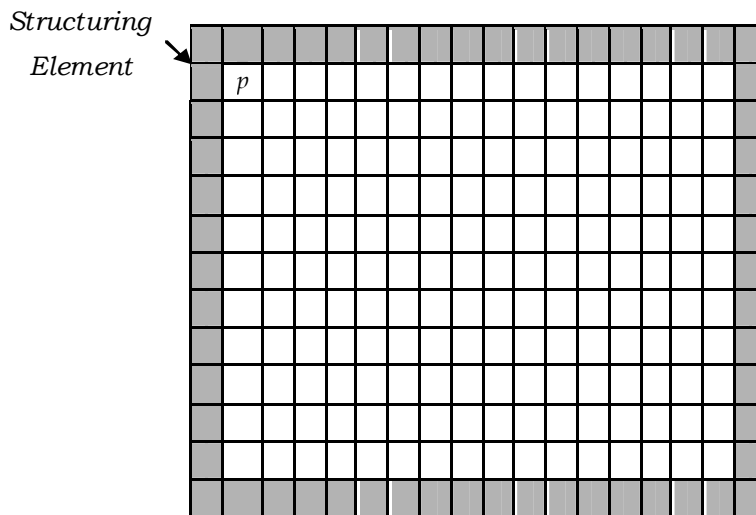
Since a 3x3 structuring element is being used, the resultant eroded image contains noise region(s) shrunk by one pixel as shown in Figure 4.8 (a) and (b). Now, if dilation is applied on the eroded image, the size of the noise region (s) will increase by one pixel, but any protruding part that has a width of one pixel will disappear completely as shown in Figure 4.8 (c).



**Figure 4.8:** Opening Operation. (a) Original image (b) Eroded Image (c) Dilated Image  
The cells in gray represent the noise region.

On the other hand, if all regions of one pixel wide are removed, this means that there are noise pixels that have been selected by the user, but which are not being taken into account during processing. However, the number of these pixels is usually very low, and will not affect the final results.

However, a problem arises when the center of the structuring element is on the border of the image: part of the structuring element will lie outside the image; hence some values inside the neighborhood are unknown. One solution is to apply the structuring element on an inner portion of the image only, discarding the border pixels. However, the resultant eroded or dilated image will be smaller than the original image. Therefore, a better solution would be to increase the size of the image by padding it with a border of one pixel (as shown in Figure 4.9), initialized with suitable values, such that when the structuring element is applied on the original image, there will always be valid data inside the structuring element. For the erosion operation, all values in the padded region are initialized to *true* while for dilation, the values are initialized to *false*. In other words, when the structuring element is positioned on a border pixel, it is in fact taking into account only pixel values inside the real image since the padded values will not affect the response.



**Figure 4.9:** Applying the structuring element on 1 pixel in an image. The pixels in white represent the real image. The pixels in gray represent the padded region.

The algorithm for performing an erosion operation on an image, using a structuring element of 3x3, is given on the next page, along with the corresponding flowchart.

- For each column  $i$  in the real image
  - For each row  $j$  in the real image
    - Search for a point in the noise bitmap with value *false* in the 3x3 neighborhood of point  $(j, i)$
    - If found, set value of point  $(j, i)$  in noise bitmap = *false*
    - Else, do not change value

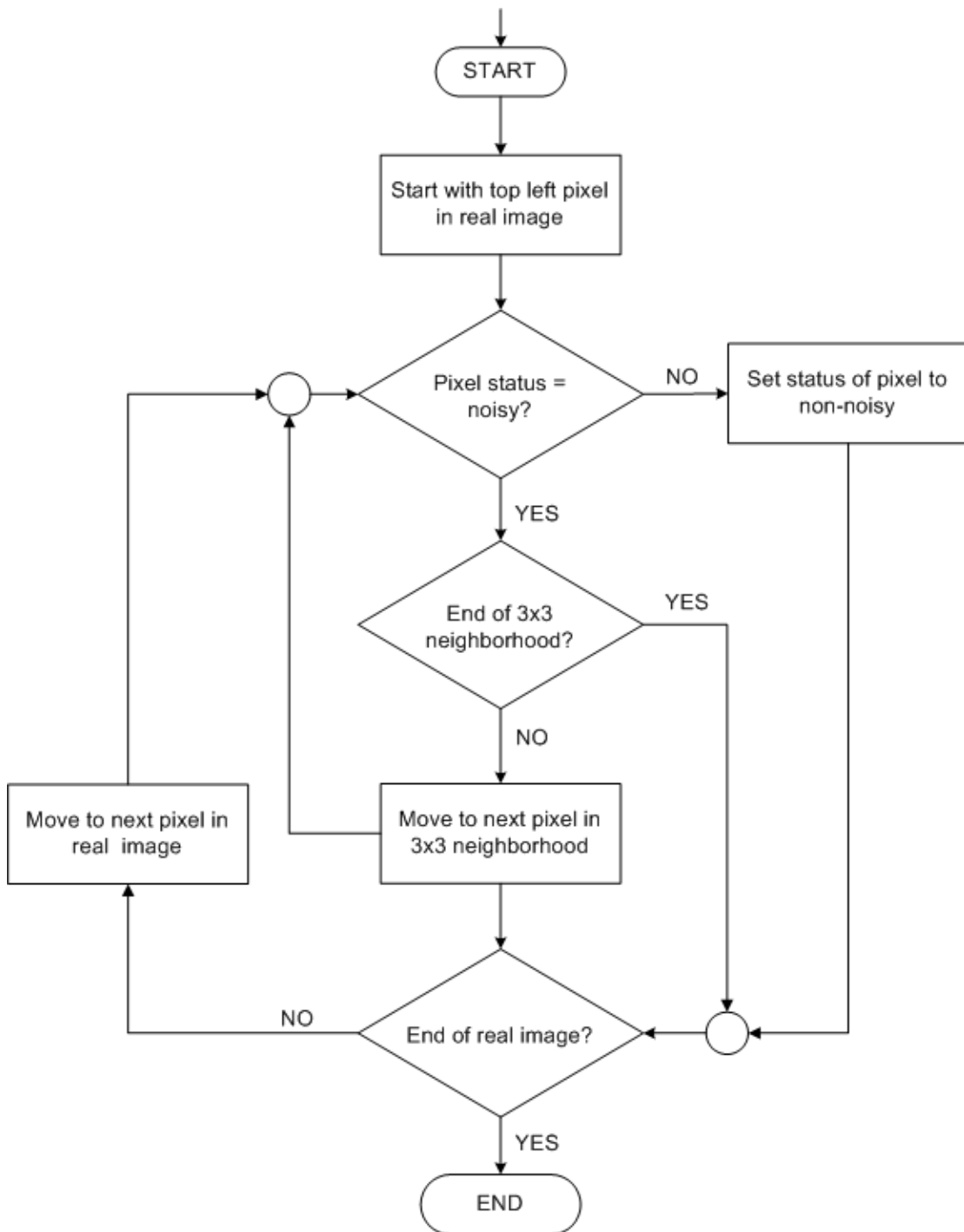


Figure 4.10: Flowchart for the erosion operation

#### 4.6.4 Step 2a

In the first part of the second step, for each point on the contour, the direction at that point needs to be computed so that a corresponding normal vector can be obtained. Based on this normal direction, an array can be built centered on the pixel so that an interpolation can be performed to restore the border of the semi-transparent blotch. The flowchart in Figure 4.11 illustrates the steps involved in step 2a.

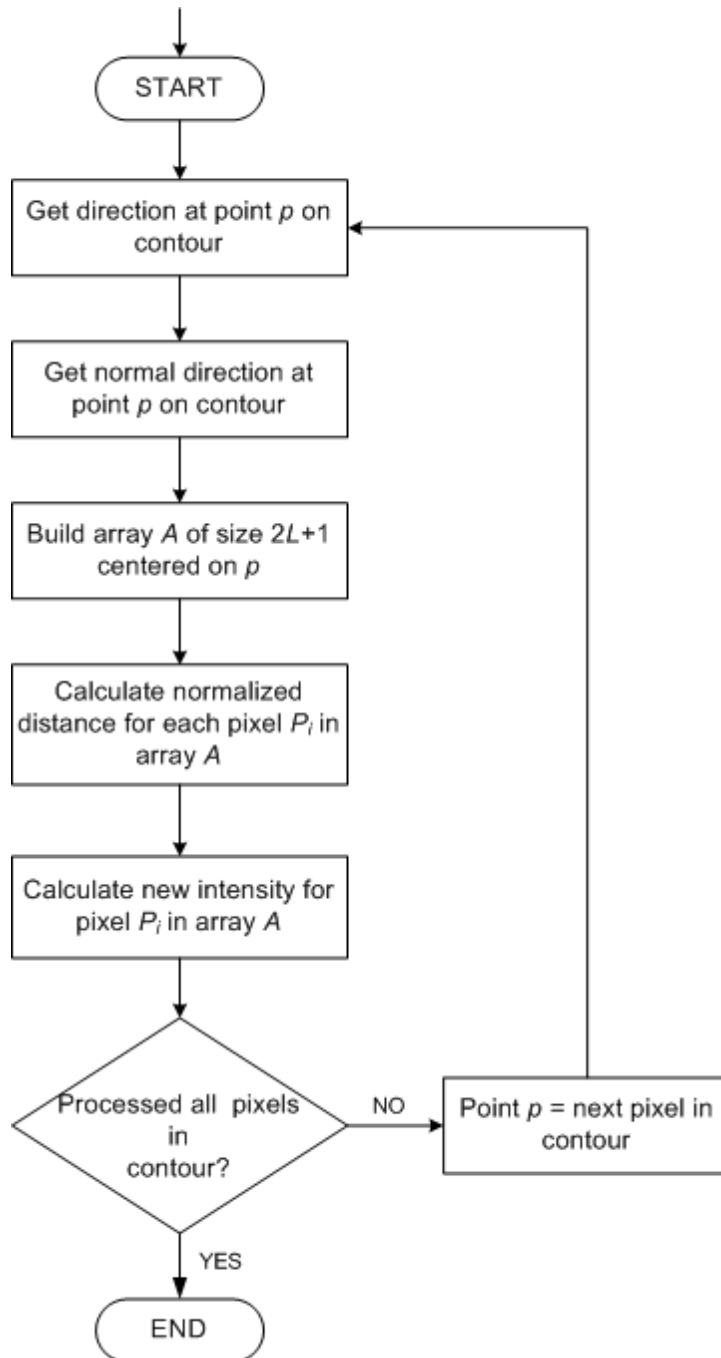
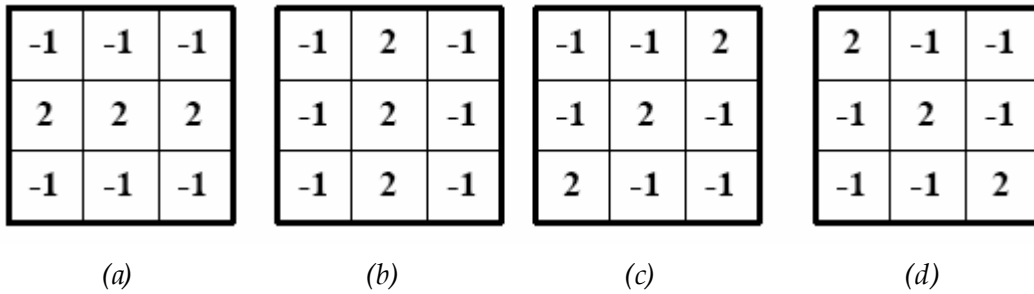


Figure 4.11: Semi-transparent blotch restoration algorithm – Step 2a

**Finding the contour direction at a point**

To find the direction at each point on the contour, four 3x3 masks have been used to approximate the direction to one of these four main orientations: horizontal, vertical, +45°, -45°. In fact, these masks are normally used for detecting lines in images. The first mask (Figure 4.12 (a)), if moved around an image, would response more strongly to lines, one pixel thick, that are oriented horizontally. With constant background, the maximum response would result when the line passes through the middle row of the mask. Similarly, the second mask (b) responds best to lines oriented vertically, while mask (c) responds best to lines oriented at 45° from the x-axis and mask (d) to lines oriented at -45°.



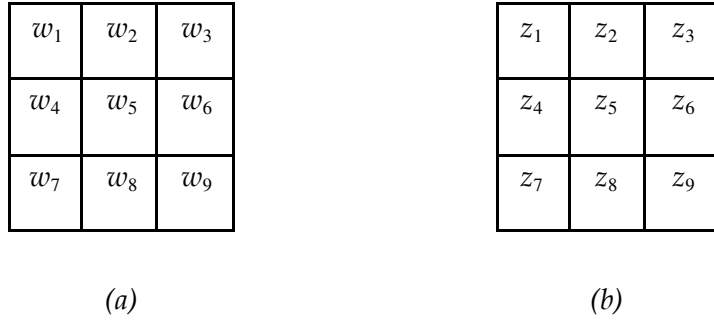
**Figure 4.12:** Masks used to determine gradient direction at a point  
 (a) Horizontal (b) Vertical (c) +45° (d) -45°

If a mask of size 3x3 is placed over any point on the contour, then the contour within this mask can be considered as a line of one pixel thick. The above four masks can thus be applied on each pixel found on the contour. The mask which generates the highest response indicates the corresponding direction. From this information, the normal direction at that point can be found: if the highest response comes from the horizontal mask, then the normal direction is vertical and vice-versa; if the highest response comes from the +45° mask, then the normal direction is -45° and vice-versa.

When a mask is applied on a point  $p$ , the response  $R_p$  at this point is calculated as follows:

$$R_p = \sum_{i=1}^{i=9} w_i z_i$$

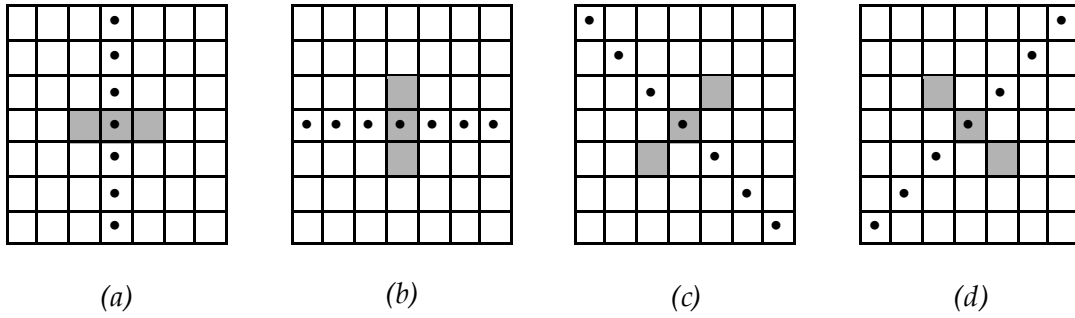
where  $w$  represents a mask and  $z$  represents the part of the image on which the mask is being applied as shown in Figures 4.13 (a) and (b).



**Figure 4.13:** Image Masks (a) Mask Coefficients (b) Image values

**Interpolation**

For each pixel  $p$  on the contour, an array of size  $2L+1$ , centered on  $p$ , is built such that  $L$  pixels of the array are inside the blotch while  $L$  pixels are outside. The array is positioned in the normal contour direction at the point  $p$ . Interpolation is a technique used to infer what the gray level value at a pixel location should be, based only on the pixel values at integer coordinate locations [GONZ92]. Here, a linear interpolation is performed between the start and the last pixel in the array, for each pixel  $P_i$  in the array, depending on the distance of the pixel under consideration. Figure 4.14 shows how the arrays are built based on the gradient direction of the contour.



**Figure 4.14:** Arrays used for interpolation based on normal gradient direction. Pixels in gray indicate the pixels found on the contour. Dotted pixels indicate the pixels used for interpolation.  $L = 3$ .

Let  $P_{start}$  denote the first pixel in the array and  $P_{end}$  denote the last one. Then the normalized distance  $d(P_i)$  of each pixel from  $P_{start}$  is given as

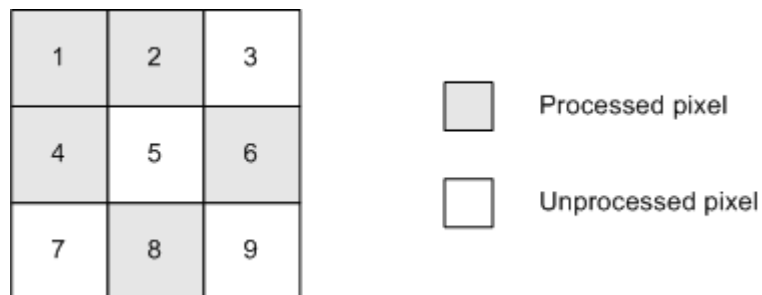
$$d(P_i) = \frac{P_i - P_{end}}{P_{start} - P_{end}}$$

The new intensity value  $\tilde{I}$  of  $P_i$  can be calculated as

$$\tilde{I}(P_i) = d(P_i) * \tilde{I}(P_{start}) + (1 - d(P_i)) * \tilde{I}(P_{end})$$

#### 4.6.5 Step 2b

The last step in restoration of semi-transparent blotches involves processing all unmodified pixels in the previous step that are within a distance  $L$  from the border of the blotch. This step also corrects any shortcoming arising from the approximation of the gradient direction in step 2a. Consequently, pixels that have already been processed needs to be labeled. For this purpose, a 2-D array, *changedpix*, is built such that the pixels that have been changed take value *true*, otherwise the value is set to *false*. Similarly to step 1, a bounding rectangular region is used to check whether a pixel is within a distance  $L$  from the border of the noise region. But in this case, the bounding rectangular region is defined by extending by a distance  $L$  the rectangle formed by the topmost, leftmost, bottommost and rightmost pixels of the noise region. For each pixel within the bounding rectangle, the corresponding value in *changepix* is checked to see if it is false. If this is the case, then a 3x3 mask centered on the pixel is applied such that the pixel value is replaced by the mean value of all already processed neighbors of the pixel. The figure below illustrates how the mean value is calculated.



**Figure 4.15:** Step 2b – 3x3 mask centered on pixel 5. The new value of pixel 5 is the mean value of pixels 1, 2, 4, 6 and 8.

## 4.7 Straight Line Restoration

Figure 4.16 shows the flowchart for the straight line restoration algorithm. The first step in this algorithm determines the coordinates of all pixels located on the boundary line by using DDA.

### DDA

The line drawing algorithm computes the pixels found on a line based from the coordinates of the two endpoints of the line. The steps in DDA are outlined below.

1. Get the 2 endpoint pixel positions  $(x_a, y_a)$  and  $(x_b, y_b)$
2. Set  $dx$  = horizontal difference between endpoint positions =  $(x_b - x_a)$   
Set  $dy$  = vertical difference between endpoint positions =  $(y_b - y_a)$
3.  $steps$  is the number of points on the line  
If  $dy > dx$ , set  $steps = dy$   
Else set  $steps = dx$
4. Determine offset needed at each step to generate the next pixel position - starting with  $(x_a, y_a)$ 
  - a)  $m = |dy/dx| = |dy| / |dx|$
  - b)  $|m| > 1$ ,  $\Delta x = \Delta y / m = \Delta y * dx / dy = dx / dy$  ( $\Delta y=1$ )  
 $xincr = \Delta x = dx / steps$
  - c)  $|m| \leq 1$ ,  $\Delta y = m(\Delta x) = \Delta x * dy / dx = dy / dx$  ( $\Delta x=1$ )  
 $yincr = \Delta y = dy / steps$
5.
  - a) If  $|m| > 1$  and  $x_a > x_b$ ,  
 $xincr = \Delta x = -dx / steps$
  - b) If  $|m| \leq 1$  and  $y_a > y_b$ ,  
 $yincr = \Delta y = -dy / steps$
6. Loop through " $steps$ " times:
  - $x = x + xincr$ ;
  - $y = y + yincr$ ;
  - If  $|m| \leq 1$ ,  $xincr=1$
  - If  $|m| > 1$ ,  $yincr=1$

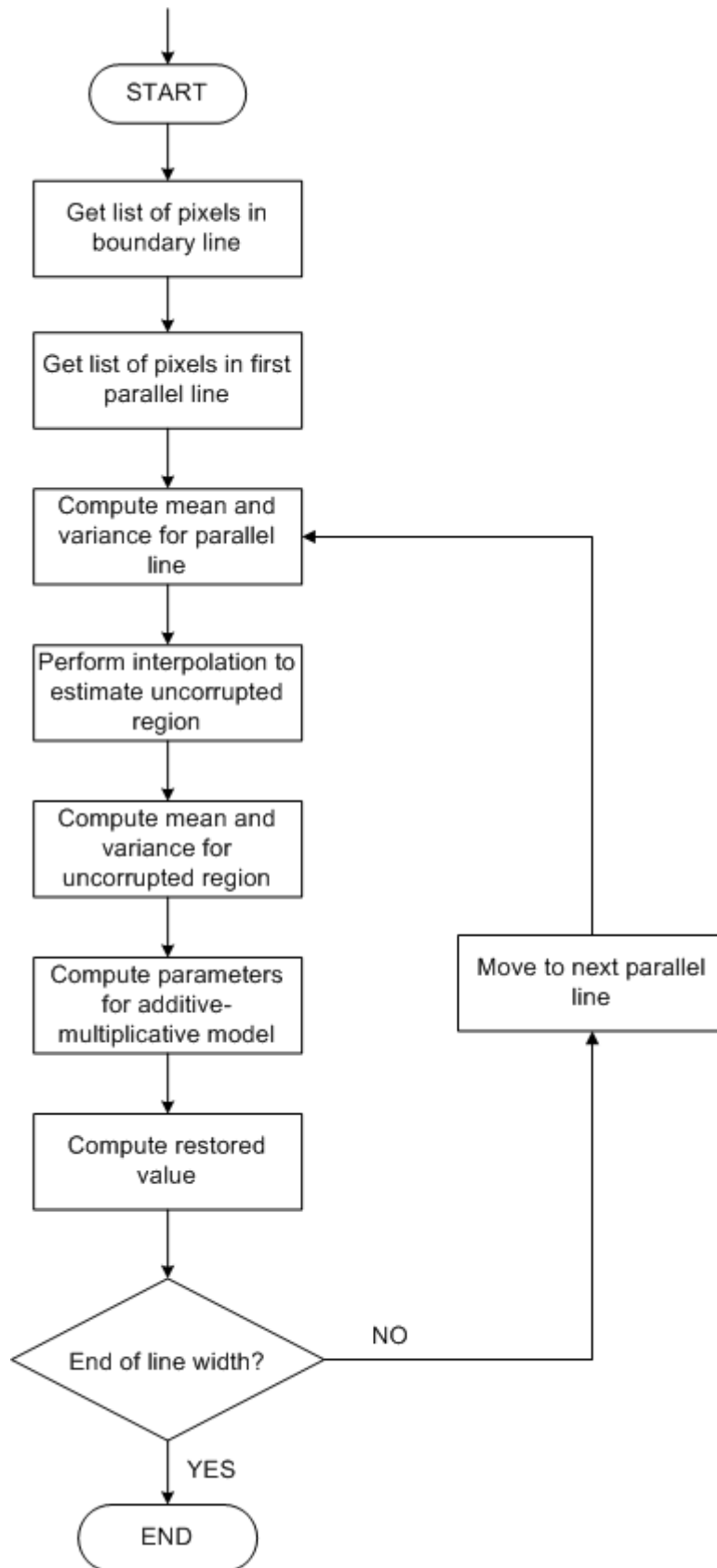


Figure 4.16: Flowchart for the straight line restoration algorithm

### Gradient Computation

Provided that the coordinates of the two endpoints of a line are available, the gradient of the line can be calculated as follows:

$$\frac{y_2 - y_1}{x_2 - x_1}$$

where  $(x_1, y_1)$  and  $(x_2, y_2)$  are the two line endpoints. However, if the line is vertical, a division by zero will occur. Therefore, the value of  $x_1$  and  $x_2$  are compared before the actual calculation is performed. If both values are the same, then the line is vertical, and a high value is associated with the gradient. This slight approximation does not affect further operations that make use of the line gradient since the gradient is only checked to see if it is greater or less than 1.

### Interpolation

To get the mean and variance of the corrupted region, the pixel intensities along a parallel line are considered. For the uncorrupted region, the values are obtained using an interpolation procedure. The direction of interpolation (horizontal or vertical) depends on the gradient of the line. If the absolute value of the gradient is less than 1, then pixels that are vertically adjacent to the boundary line (Figure 4.17) are considered – here a vertical interpolation will be performed, otherwise pixels that are horizontally adjacent to the boundary line are considered. In any case, the number of pixels making up the parallel line will be the same as that of any one of the boundary line and the number of parallel lines to be processed depends on the line width.

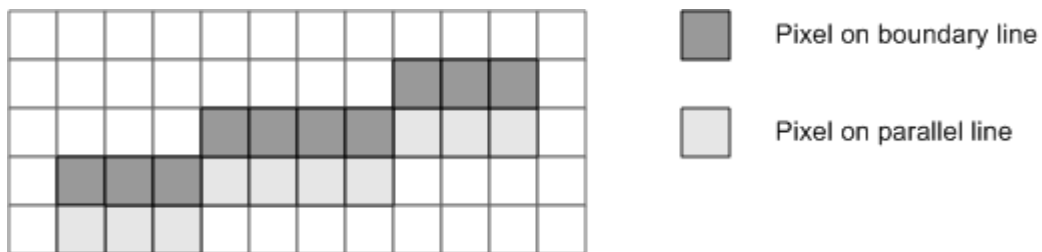


Figure 4.17: Vertically adjacent pixels

The steps for applying the additive-multiplicative model in the straight line restoration procedure are outlined below.

For each parallel line  $L_i$  where  $1 < i < \text{line width}$

- If line gradient  $< 1$ , add  $i$  to each  $y$ -coordinate of boundary line to get coordinates of each pixel in  $L_i$ . Else, add  $i$  to the  $x$ -coordinates.
- Find mean and variance of pixels in  $L_i$ .
- If line gradient  $< 1$ , perform vertical interpolation. Else, perform horizontal interpolation.
- Find estimated mean and variance for uncorrupted region.
- Compute  $\alpha$  and  $\beta$ , the parameters for the additive-multiplicative model.
- Compute restored value using equation (3-4).

## 4.8 Image Noise Removal Method

### 4.8.1 Basic Method

The flowchart shown in Figure 4.18 shows the steps involved in the basic algorithm of the image noise removal method. It is the user who specifies the number of iterations required. For the initial iteration, the repair subimage needs to be multiplied by the noise bitmap. Since the noise bitmap values are either *true* (noisy pixel) or *false* (non-noisy pixel), the intensity value of all noisy pixels within the repair mask is set to *true*.

#### Repair and Sample Subimages

Since the number of repair or sample subimages that the user will specify is not known in advance, two linked lists are used: one to store repair subimages, while the other one stores sample subimages. For each subimage, the coordinates of its four vertices are stored as well as the magnitude values after the Fourier transform is applied to the subimage.

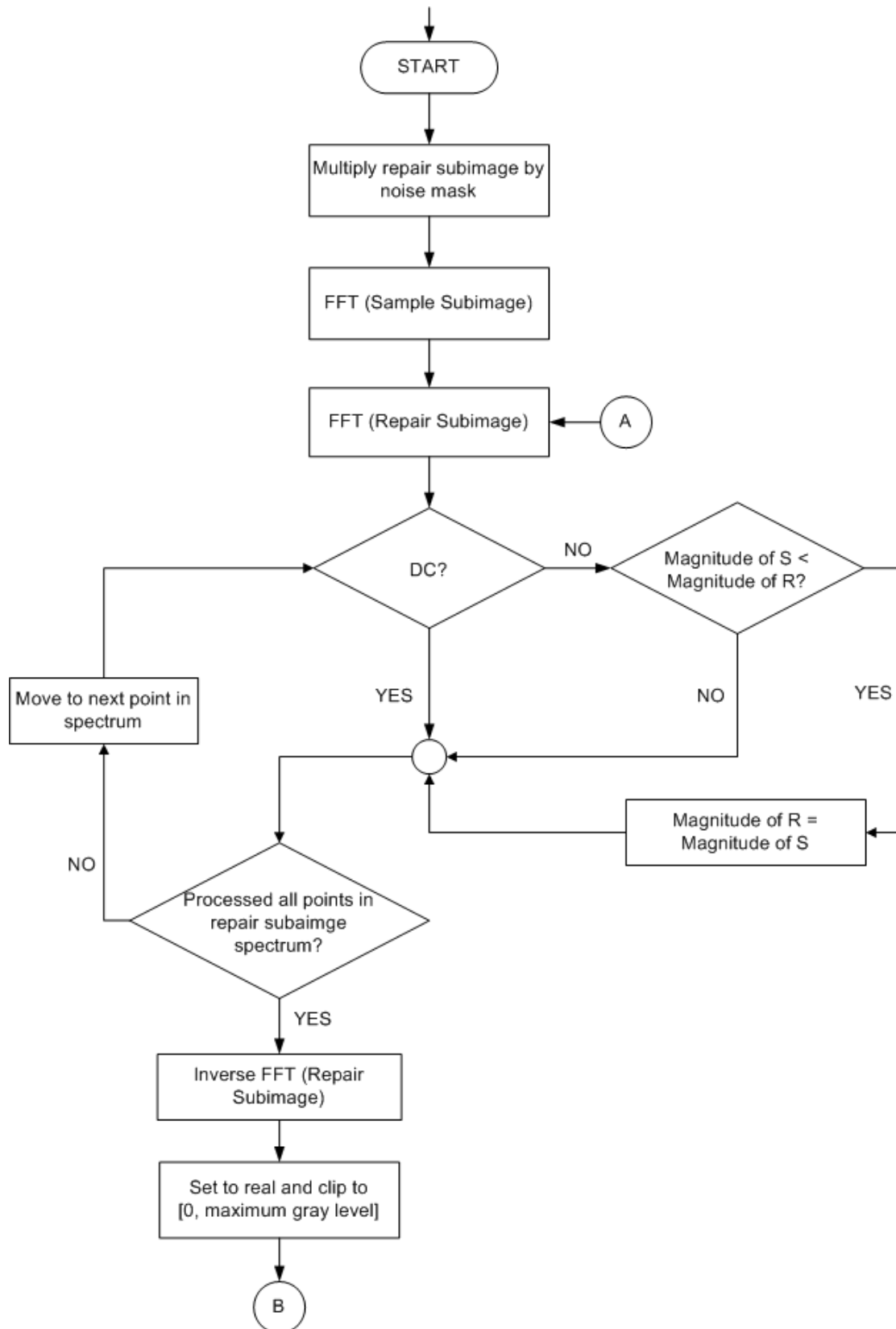


Figure 4.18: Flowchart for the basic algorithm

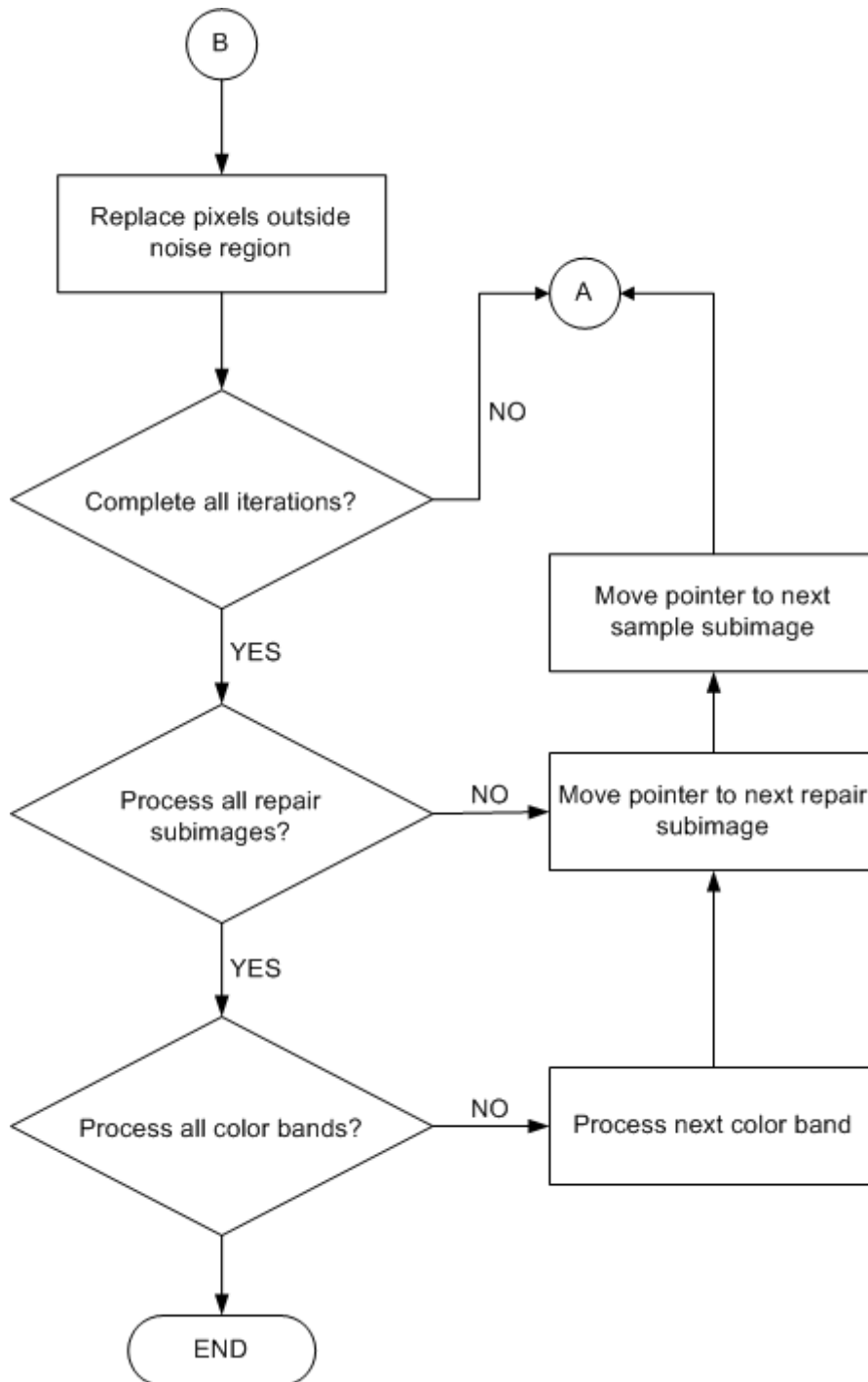
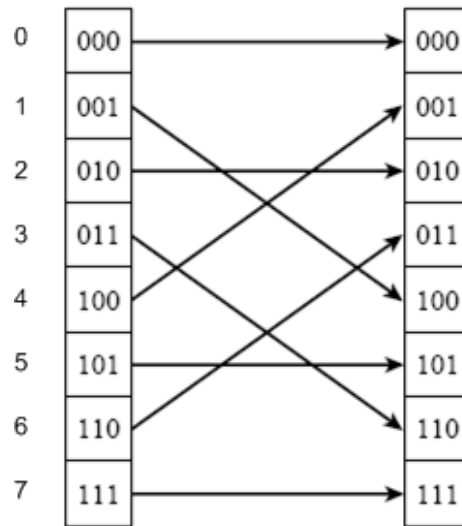


Figure 4.18 (continued)

## FFT

The implementation of the FFT is done in two parts. The first part sorts the data using the bit reversal rule so that further computation can be done efficiently. The second part calculates transforms of length 2, 4, 8, ...,  $M$ , where  $M$  is the length of the data set. The **bit reversal** rule is used to reorder the input of the data set. If  $x$  represents any valid argument value in  $f(x)$ , the corresponding argument in the reordered array is obtained by expressing  $x$  in binary and (left-to-right) reversing the bits. For example, 6 (110) is the bit reverse of 3 (011) and vice-versa. Consequently, the two values can be swapped. Figure 4.19 shows an example of bit reversal on an array of size 8.



**Figure 4.19:** Reordering an array of size 8 using the bit reversal rule

Since the FFT implemented is derived from equations (3-13) and (3-14) where it has been assumed that  $M = 2^n$ , this means that only images with a height and width that can be expressed in powers of two can be used in this implementation of FFT. Therefore, before the actual FFT algorithm is applied on an image (or subimage), the height and width of the image is checked to see if they are powers of two. If this is not the case, then the data set that is passed as parameter to the FFT is padded with zeroes up to the next power of two.

The input array to the function *FFT* in the class *Image* is a 1-D array that starts from index 1, and holds complex numbers. The odd indices hold real data and the even indices hold the imaginary part. On the other hand, the data that needs to be transformed are the intensity values of the image i.e. the values are found in a 2-D

array (we can discard the third dimension of the array since the transform is applied to only one band at a time). Thus, this data should be copied to a 1-D array of size  $(newheight * newwidth * 2 + 1)$ , where *newheight* and *newwidth* are the height and width values of the image if these values are powers of two; otherwise, the values are converted to the next power of two. We add 1 to the size because the array starts at index 1. The size of the image is multiplied by 2 because the data array is actually holding complex data points. Since the intensity values of the image before it is transformed are all real, only the odd indices are filled with these values, the other array slots are set to zero. After the FFT procedure, the 1-D array passed as input is filled with the transformed complex values in the same way.

The transform needs to be centered so that the DC lies at the centre of the transformed image. The image is thus multiplied by  $(-1)^{x+y}$  before copying it to the 1-D array. Also, the same function is used for FFT and inverse FFT, since the only difference between the two operations is the sign of *j* in the exponential of equation (3-9). Thus, to perform forward FFT, the parameter *isign* of function *FFT* is set to 1 and for inverse FFT, *isign* is set to -1. Also for inverse FFT, the values obtained after calling function *FFT* need to be divided by the total number of complex points in the data set – this represents the  $1/M$  in front of equation (3-9).

The pseudo-code for forward FFT (on one color band) is as follows:

1. Check if height and width of image =  $2^n$ , otherwise set number of columns and rows of image to next  $2^n$
2. For each column *i* in the image
  - For each row *j* in the image
    - Set intensity value at point (*j*, *i*) to 0 if beyond actual height or width of image
    - Else, multiply intensity value at point (*j*, *i*) in image by  $(-1)^{x+y}$
3. Create 1-D array *data* of size  $(newheight * newwidth * 2 + 1)$ .
4. Declare a variable *k*, with initial value  $k = 1$
5. For each column *i* in the image
  - For each row *j* in the image
    - $data [k] = \text{intensity value at point } (j, i)$
    - $data [k + 1] = 0$

- $k = k + 2$
- 6. Call function FFT with parameter *isign* set to 1
- 7. Extract transformed values from 1-D array to 2-D complex array

### Comparing magnitudes of repair and sample subimage transforms

Since the transform has been centered over the image, the lowest frequency is found at the centre of the image. Therefore, all other frequencies in the repair and sample subimages are compared to reshape the spectrum of the repair subimage. The steps for this process is given below.

For each column  $v$  in the spectrum of the repair subimage

- For each row  $u$  in the spectrum of the repair subimage
  - If NOT ( $v = \text{height of subimage}/2$  AND  $u = \text{width of subimage}/2$ )
    - If magnitude at  $(u, v)$  in repair subimage  $>$  magnitude at  $(u, v)$  in sample subimage
      - Magnitude at  $(u, v)$  in repair subimage = magnitude at  $(u, v)$  in sample subimage
    - Else, magnitude of repair subimage is not modified

### Clipping intensity values

The following pseudo-code outlines the clipping process in the spatial domain:

For each pixel in the image (in one band only)

- If pixel intensity  $>$  maximum graylevel, set pixel intensity = maximum graylevel
- If pixel intensity  $< 0$ , set pixel intensity = 0

### Replacement Process

Before a repair subimage is processed, its original values are copied so that all subsequent iterations that use this repair subimage can perform the replacement procedure. The steps in the replacement procedure are given on the next page.

For each column  $i$  in the repair subimage

- For each row  $j$  in the repair subimage
  - If pixel  $(j, i)$  is a non-noisy pixel, replace pixel  $(j, i)$  of the repair subimage by its original value
  - Else do not replace the pixel value

It should be highlighted in the above steps that if the pixel value is not replaced, then the value obtained after processing in the frequency domain is used in the next iteration.

### 4.8.2 Soft Scratch Method

The process of determining whether a pixel forms part of the soft edged region of width  $d$  is similar to the problem of finding the pixels to estimate the uncorrupted region in the semi-transparent blotch restoration method (see section 4.6). Therefore, in this case too, the minimum distance between a pixel in the repair subimage (outside the noise region) and the pixels on the border of the noise region needs to be found. If this distance is less or equal to  $d$ , then the pixel is part of the soft edged region. If the distance is greater, then the pixel is outside the soft edged region and  $w$  (in equation 3-15) is set to 1. If a pixel within the repair subimage is part of the noise region, then  $w$  is set to 0.

To allocate a suitable value to  $w$  when a pixel is within the soft edged region, a linear interpolation is performed. If  $x_d$  represents the minimum distance of the pixel  $x$  from the border of the noise region, then the value of  $w$  is given as:

$$w = \frac{x_d}{d}$$

Since

$$0 < x_d < d$$

then

$$0 < w < 1$$

such that  $w$  increases towards 1 as the pixel is further away from the noise region.

In the soft scratch method, the border of the noise region is needed only to speed further processing. However, only one border is defined given the topmost pixel in

the whole noise region, even if this region comprises of several smaller regions. Thus, only one noise region would be processed meaning that the user can select only one noise. Therefore, to conciliate speed and robustness, the algorithm counts the number of noise regions the user has selected. If this number is exactly one, then the pixels on the border of the noise region are used in consequent processing, else if there are more than one noise region, the whole noise region is used.

### Counting the number of noise regions

The algorithm for counting the number of noise regions mainly scans the noise bitmap and labels each separate region found. The number of distinct labels assigned is then the number of regions present. The algorithm is given below.

1. Build the noise bitmap
2. Declare a 2-D array *labelimg* of the same size as the noise bitmap to hold label values
3. Set all values of *labelimg* to -1
4. Declare a variable *ID*, with an initial value of 0, to keep track of label values being assigned
5. For each column *i* in the noise bitmap
  - For each row *j* in the noise bitmap
    - If point (*j, i*) is a noise pixel AND labeled value = -1
      - Set value of point (*j, i*) in *labelimg* = *ID*
      - Set value of all noise pixels connected to point (*j, i*) = *ID*
      - Increment *ID* by 1
6. Number of regions = value of *ID*

4-connectivity has been used to find all noise pixels connected to a certain pixel, just like in the automatic noise selection method (section 4.4).

The pseudo-code for the replacement procedure in the soft scratch method is as follows:

- For each pixel  $p$  inside the repair subimage
  - If  $p$  is a noisy, set  $w = 0$
  - Else
    - If number of noise regions = 1, find minimum distance between  $p$  and border pixels
    - Else, find minimum distance between  $p$  and all noise pixels
    - If minimum distance > width of soft edged region, set  $w = 1$
    - Else, set  $w = (\text{minimum distance} / \text{width of soft edged region})$
  - New value of  $p = [(1-w) * \text{current value of } p] + [w * \text{original value of } p]$

### 4.8.3 Split Frequency Method

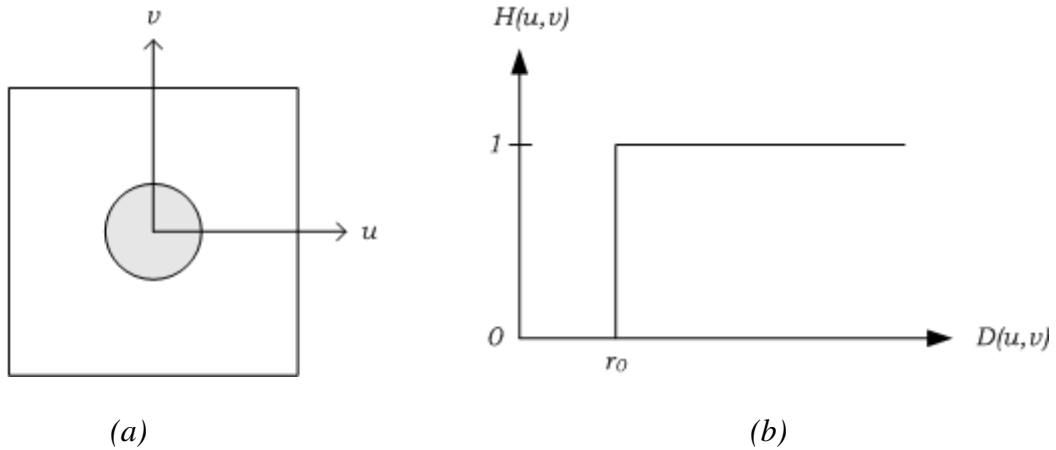
After the Fourier transform of the repair subimage has been obtained, a high pass filter needs to be used to separate the lower frequencies below a cut-off frequency  $r_0$  in the transform from the higher frequencies. Usually in high pass filtering, a filter function  $H(u,v)$  is applied to the real and the imaginary parts of the transform  $F(u,v)$ . The Fourier transform of the output image,  $G(u,v)$ , is then defined as

$$G(u,v) = H(u,v) F(u,v)$$

where

$$H(u,v) = \begin{cases} 0 & \text{if } D(u,v) \leq r_0 \\ 1 & \text{if } D(u,v) > r_0 \end{cases}$$

$D(u,v)$  is the distance of a point from the origin in the spectrum such that  $D(u,v) = \sqrt{u^2 + v^2}$ . Frequencies below  $r_0$  are therefore suppressed as shown in Figure 4.20. For the split frequency method however, the low frequencies are not suppressed: they are temporarily stored, since we need to merge these frequencies with the processed high frequencies later.



**Figure 4.20:** High pass filtering (a) 2-D plot of ideal highpass filter. The shaded region inside the circle indicates suppressed frequencies. (b) Radial cross section of ideal highpass filter

In addition to the original repair subimage values that need to be copied before processing is done on the subimage, the high pass filtered subimage values need to be stored during each iteration so that they be used in the next iteration. These values are used in a replacement operation that takes place just after the magnitudes of the repair and sample subimages have been compared. The steps for this replacement procedure are exactly the same as in the soft scratch method except the last step which becomes:

New value of  $p$  =

$$[ (1-w) * \text{current value of } p ] + [ w * \text{value of } p \text{ in previous iteration} ]$$

for each  $p$  inside the high pass filtered subimage.

## 4.9 Summary

In this chapter, the implementation details of the three techniques proposed have been given. The problems encountered while implementing the design and the solutions that have been used are also described.