Image Processing Application

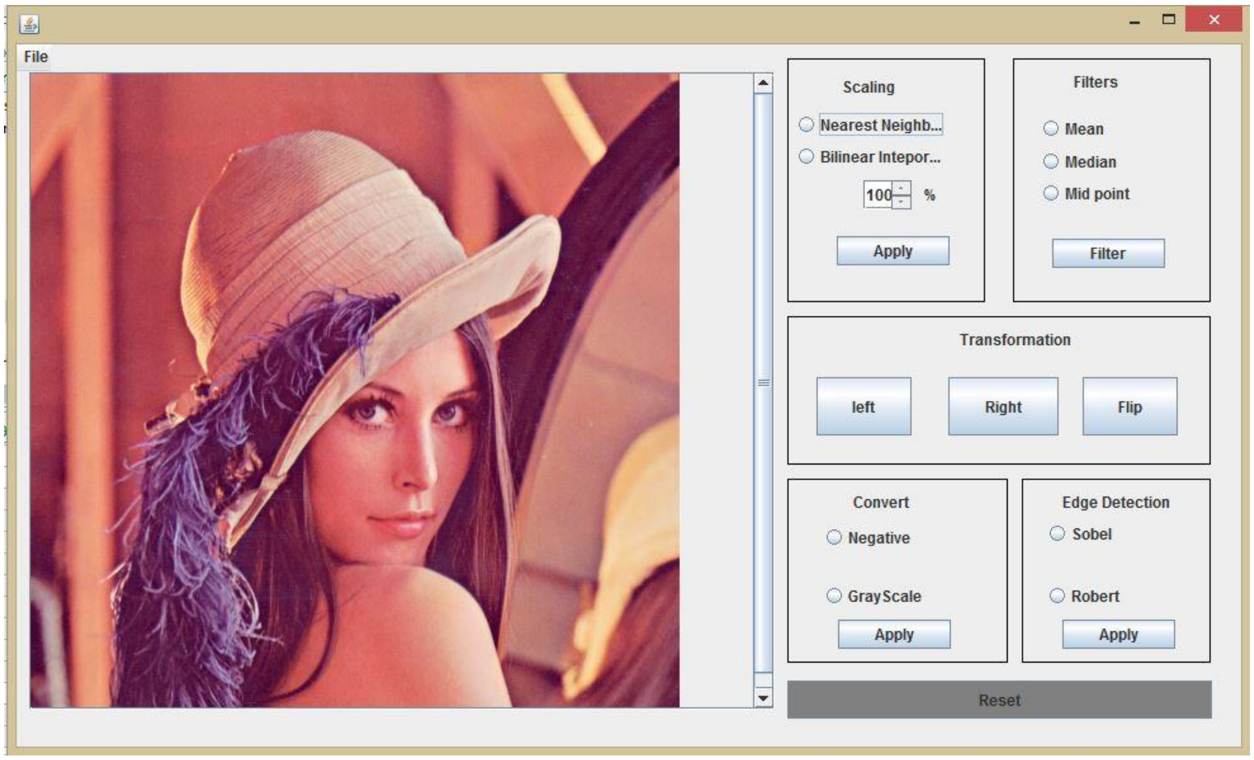
N.M.S.Nadeeshan

(140405J)

Introduction

This is a Java application which is capable of handling care theories which were completed in the Image Processing class. The application takes a digital images as an input and let the user to modify and save it. The main functions will be

- Image scaling
- Filtering
- Edge detection
- Getting the negative, grayscale image
- Transformation

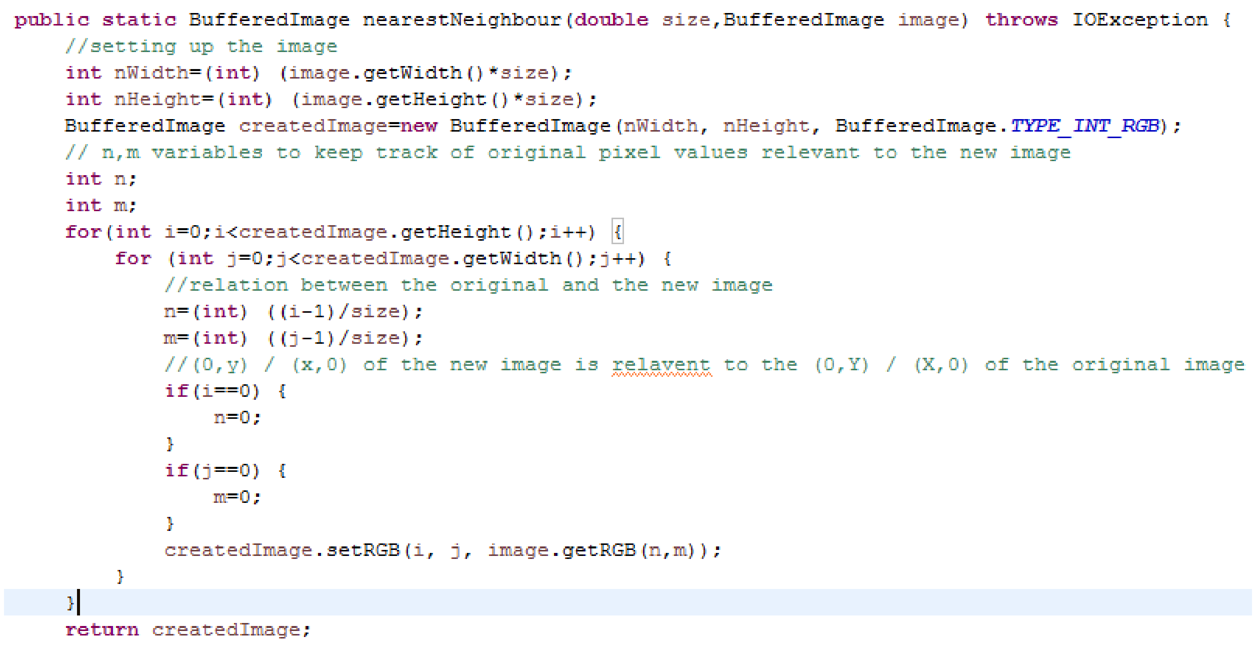


Functions

Resampling

Nearest Neighbour

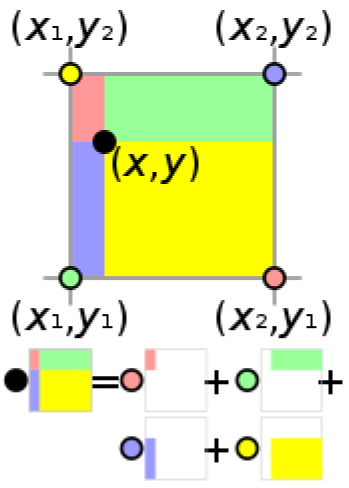
This is the one of the basics operations in the image processing operations. The user can input the scale that he wants. Here a new blank image will be created for the new width and the height. The pixels will be coloured by using the colours of the nearby pixels. Here is the code for the method.



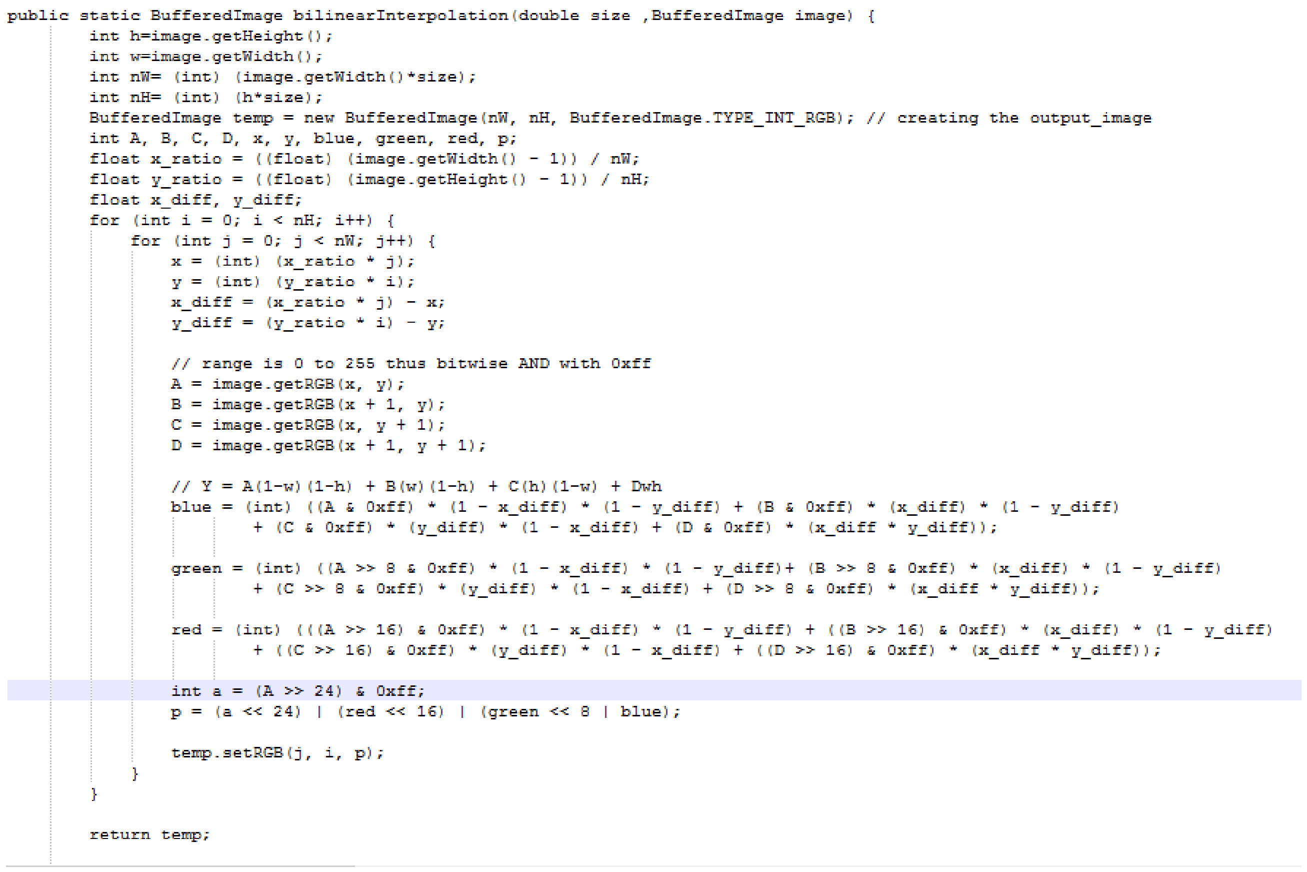
```java
public static BufferedImage nearestNeighbour(double size,BufferedImage image) throws IOException {
    //setting up the image
    int nWidth=(int) (image.getWidth()*size);
    int nHeight=(int) (image.getHeight()*size);
    BufferedImage createdImage=new BufferedImage(nWidth, nHeight, BufferedImage.TYPE_INT_RGB);
    // n,m variables to keep track of original pixel values relevant to the new image
    int n;
    int m;
    for(int i=0;i<createdImage.getHeight();i++) {
        for (int j=0;j<createdImage.getWidth();j++) {
            //relation between the original and the new image
            n=(int) ((i-1)/size);
            m=(int) ((j-1)/size);
            //(0,y) / (x,0) of the new image is relavent to the (0,Y) / (X,0) of the original image
            if(i==0) {
                n=0;
            }
            if(j==0) {
                m=0;
            }
            createdImage.setRGB(i, j, image.getRGB(n,m));
        }
    }
    return createdImage;
```

Bi-linear Interpolation

This is a more advanced way of resampling an image. The basic concept of linear of interpolation is used in an advanced way to estimate the pixel values of the newly created image. The bi-linear interpolation is a combination of three linear interpolation. First interpolate (x1,y2) and (x2,y2) and then (x1,y1) and (x2,y1) then interpolate the two resulting values in order to get the pixel value of the (x,y).



Here is the implementation.



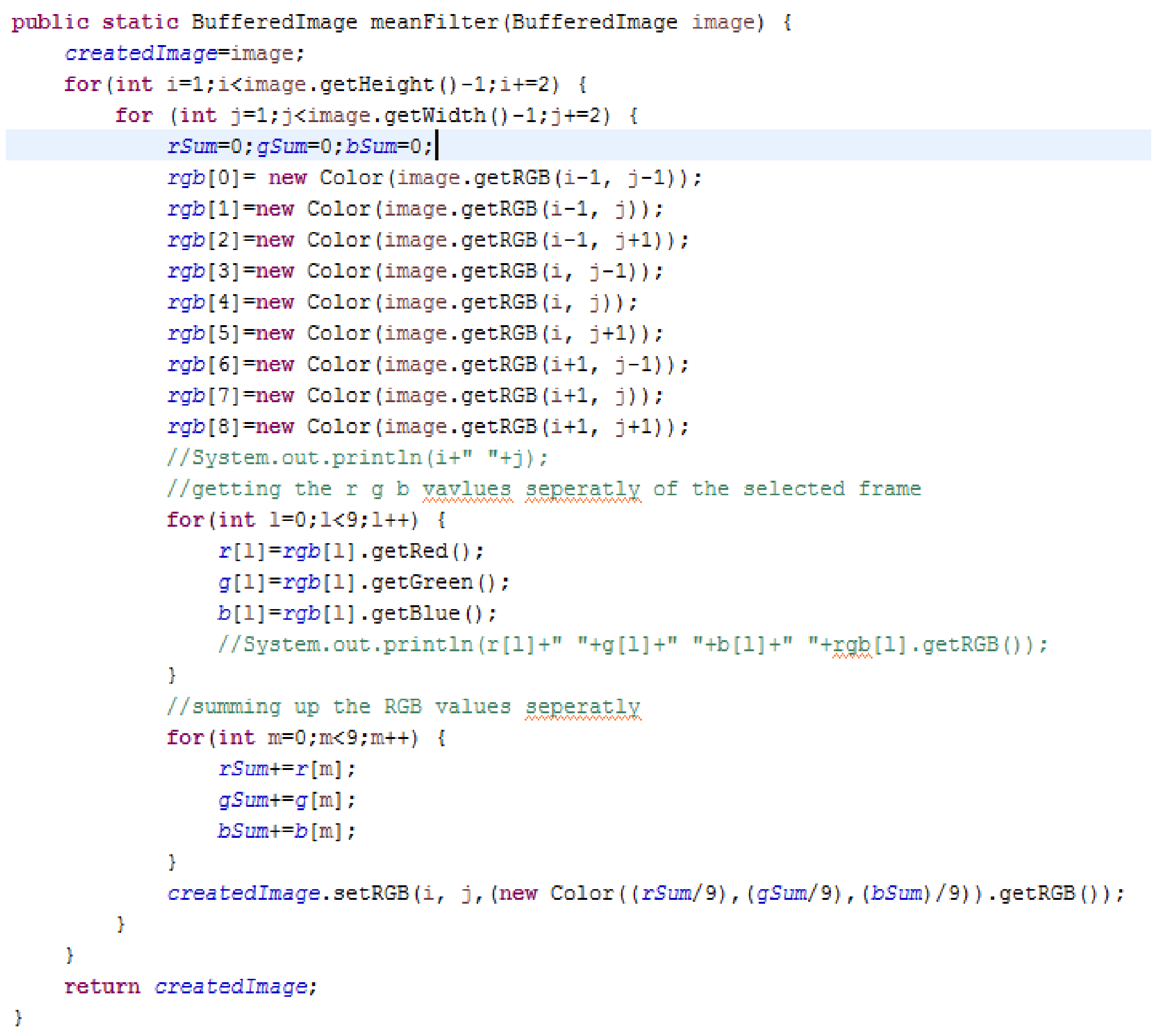
```
public static BufferedImage bilinearInterpolation(double size ,BufferedImage image) {
        int h=image.getHeight();
        int w=image.getWidth();
        int nW= (int) (image.getWidth()*size);
        int nH= (int) (h*size);
        BufferedImage temp = new BufferedImage(nW, nH, BufferedImage.TYPE_INT_RGB); // creating the output_image
        int A, B, C, D, x, y, blue, green, red, p;
        float x_ratio = ((float) (image.getWidth() - 1)) / nW;
        float y_ratio = ((float) (image.getHeight() - 1)) / nH;
        float x_diff, y_diff;
        for (int i = 0; i < nH; i++) {
            for (int j = 0; j < nW; j++) {
                x = (int) (x_ratio * j);
                y = (int) (y_ratio * i);
                x_diff = (x_ratio * j) - x;
                y_diff = (y_ratio * i) - y;

                // range is 0 to 255 thus bitwise AND with 0xff
                A = image.getRGB(x, y);
                B = image.getRGB(x + 1, y);
                C = image.getRGB(x, y + 1);
                D = image.getRGB(x + 1, y + 1);

                // Y = A(1-w)(1-h) + B(w)(1-h) + C(h)(1-w) + Dwh
                blue = (int) ((A & 0xff) * (1 - x_diff) * (1 - y_diff) + (B & 0xff) * (x_diff) * (1 - y_diff)
                        + (C & 0xff) * (y_diff) * (1 - x_diff) + (D & 0xff) * (x_diff * y_diff));

                green = (int) ((A >> 8 & 0xff) * (1 - x_diff) * (1 - y_diff)+ (B >> 8 & 0xff) * (x_diff) * (1 - y_diff)
                        + (C >> 8 & 0xff) * (y_diff) * (1 - x_diff) + (D >> 8 & 0xff) * (x_diff * y_diff));

                red = (int) (((A >> 16) & 0xff) * (1 - x_diff) * (1 - y_diff) + ((B >> 16) & 0xff) * (x_diff) * (1 - y_diff)
                        + ((C >> 16) & 0xff) * (y_diff) * (1 - x_diff) + ((D >> 16) & 0xff) * (x_diff * y_diff));

                int a = (A >> 24) & 0xff;
                p = (a << 24) | (red << 16) | (green << 8 | blue);

                temp.setRGB(j, i, p);
            }
        }

        return temp;
```

Noise Reduction

Noise reduction is one of the most using image manipulation technology in image processing. Noise can be random or white noise with no coherence, or coherent noise introduced by the device's mechanism or processing algorithms. There are 3 main filtering algorithms to filter those noises from an image. We use a mask as the filter which filters the image.

Mean Filter

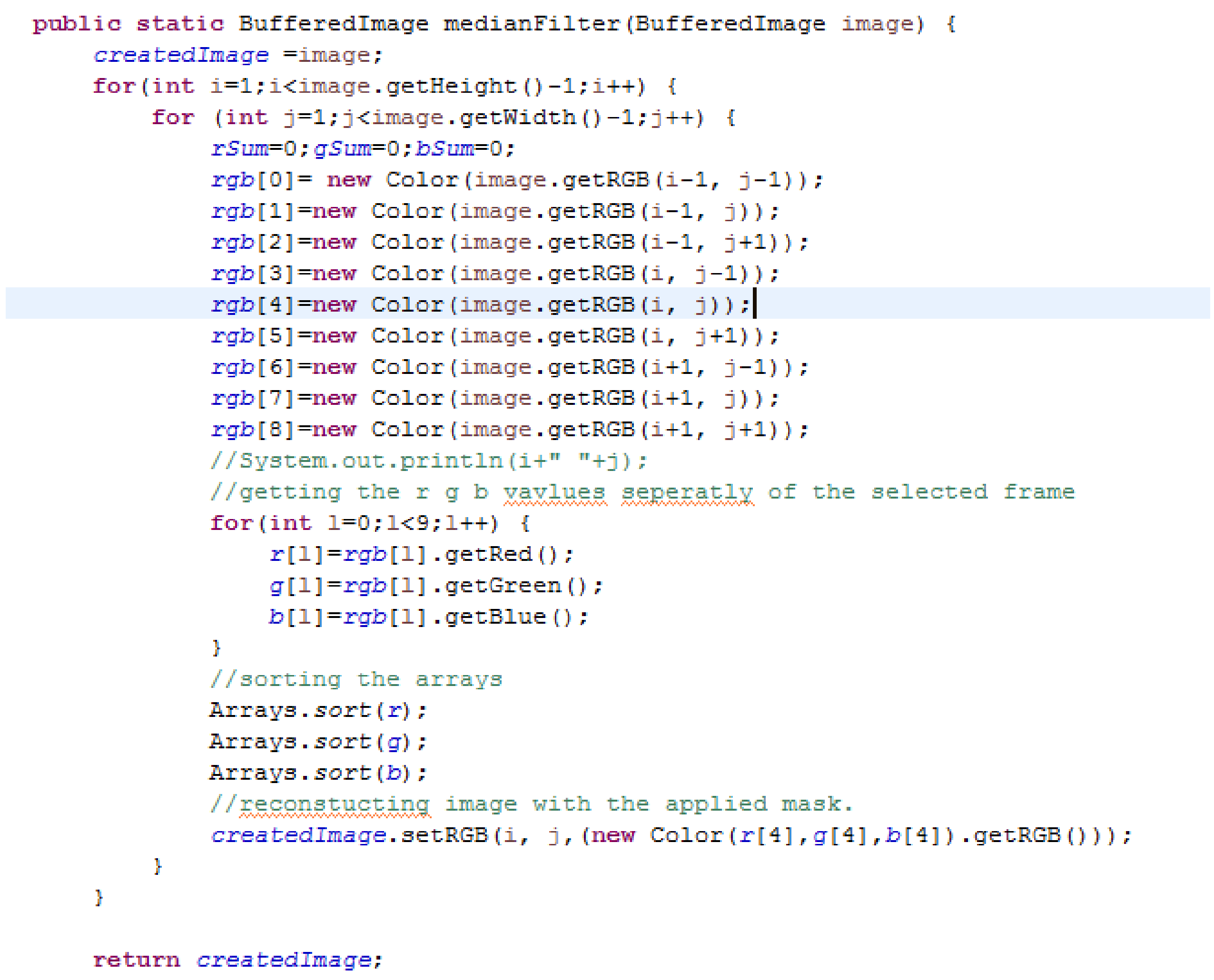
This filter has a mask [[1/9,1/9,1/9], [1/9,1/9,1/9], [1/9,1/9,1/9]] . This results a blurring the image. This mask averages the colours of the 9 pixels around the selected pixel. Filter code is as follows.



```java
public static BufferedImage meanFilter(BufferedImage image) {
    createdImage=image;
    for(int i=1;i<image.getHeight()-1;i+=2) {
        for (int j=1;j<image.getWidth()-1;j+=2) {
            rSum=0;gSum=0;bSum=0;
            rgb[0]= new Color(image.getRGB(i-1, j-1));
            rgb[1]=new Color(image.getRGB(i-1, j));
            rgb[2]=new Color(image.getRGB(i-1, j+1));
            rgb[3]=new Color(image.getRGB(i, j-1));
            rgb[4]=new Color(image.getRGB(i, j));
            rgb[5]=new Color(image.getRGB(i, j+1));
            rgb[6]=new Color(image.getRGB(i+1, j-1));
            rgb[7]=new Color(image.getRGB(i+1, j));
            rgb[8]=new Color(image.getRGB(i+1, j+1));
            //System.out.println(i+" "+j);
            //getting the r g b vavlues seperatly of the selected frame
            for(int l=0;l<9;l++) {
                r[l]=rgb[l].getRed();
                g[l]=rgb[l].getGreen();
                b[l]=rgb[l].getBlue();
                //System.out.println(r[l]+" "+g[l]+" "+b[l]+" "+rgb[l].getRGB());
            }
            //summing up the RGB values seperatly
            for(int m=0;m<9;m++) {
                rSum+=r[m];
                gSum+=g[m];
                bSum+=b[m];
            }
            createdImage.setRGB(i, j,(new Color((rSum/9),(gSum/9),(bSum)/9)).getRGB());
        }
    }
    return createdImage;
}
```

Median Filter

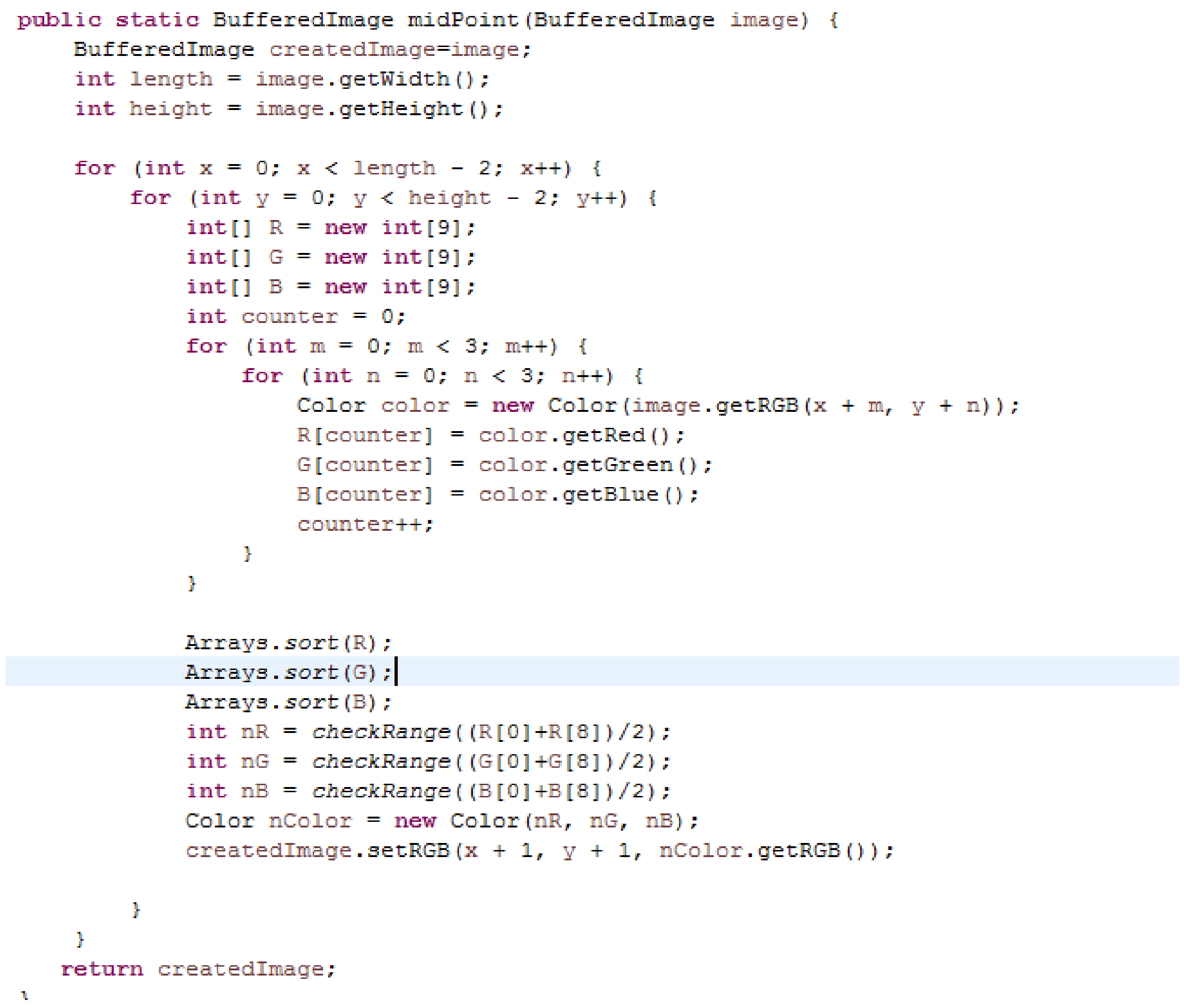
This contains a mask with all 1's and the pixel will be replaced by the median value of the resulting colour array. The median filter also takes 9 pixels around the selected pixel as the input.



```java
public static BufferedImage medianFilter(BufferedImage image) {
    createdImage =image;
    for(int i=1;i<image.getHeight()-1;i++) {
        for (int j=1;j<image.getWidth()-1;j++) {
            rSum=0;gSum=0;bSum=0;
            rgb[0]= new Color(image.getRGB(i-1, j-1));
            rgb[1]=new Color(image.getRGB(i-1, j));
            rgb[2]=new Color(image.getRGB(i-1, j+1));
            rgb[3]=new Color(image.getRGB(i, j-1));
            rgb[4]=new Color(image.getRGB(i, j));
            rgb[5]=new Color(image.getRGB(i, j+1));
            rgb[6]=new Color(image.getRGB(i+1, j-1));
            rgb[7]=new Color(image.getRGB(i+1, j));
            rgb[8]=new Color(image.getRGB(i+1, j+1));
            //System.out.println(i+" "+j);
            //getting the r g b vavlues seperatly of the selected frame
            for(int l=0;l<9;l++) {
                r[l]=rgb[l].getRed();
                g[l]=rgb[l].getGreen();
                b[l]=rgb[l].getBlue();
            }
            //sorting the arrays
            Arrays.sort(r);
            Arrays.sort(g);
            Arrays.sort(b);
            //reconstucting image with the applied mask.
            createdImage.setRGB(i, j,(new Color(r[4],g[4],b[4]).getRGB()));
        }
    }

    return createdImage;
```

Mid Point Filter

In the midpoint filter, the color value of each pixel is replaced with the average of maximum and minimum of the selected 9 pixel around the selected pixel.



```java
public static BufferedImage midPoint(BufferedImage image) {
    BufferedImage createdImage=image;
    int length = image.getWidth();
    int height = image.getHeight();

    for (int x = 0; x < length - 2; x++) {
        for (int y = 0; y < height - 2; y++) {
            int[] R = new int[9];
            int[] G = new int[9];
            int[] B = new int[9];
            int counter = 0;
            for (int m = 0; m < 3; m++) {
                for (int n = 0; n < 3; n++) {
                    Color color = new Color(image.getRGB(x + m, y + n));
                    R[counter] = color.getRed();
                    G[counter] = color.getGreen();
                    B[counter] = color.getBlue();
                    counter++;
                }
            }

            Arrays.sort(R);
            Arrays.sort(G);
            Arrays.sort(B);
            int nR = checkRange((R[0]+R[8])/2);
            int nG = checkRange((G[0]+G[8])/2);
            int nB = checkRange((B[0]+B[8])/2);
            Color nColor = new Color(nR, nG, nB);
            createdImage.setRGB(x + 1, y + 1, nColor.getRGB());

        }
    }
   return createdImage;
}
```

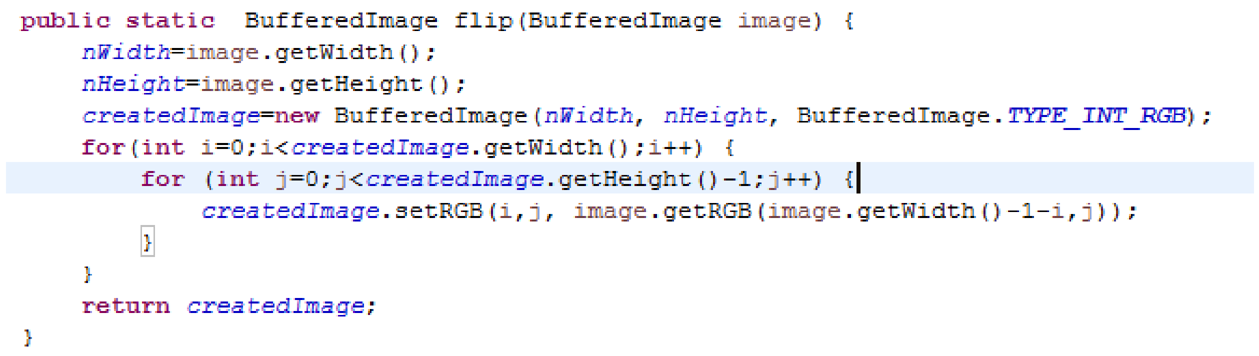
Point Operation

The point operations are done to modify the intensity of individual pixels without depending on other pixels (either neighborhood pixels). There are two types of point operations.

- Homogeneous point operations
- Heterogeneous point operations

Vertical Flip

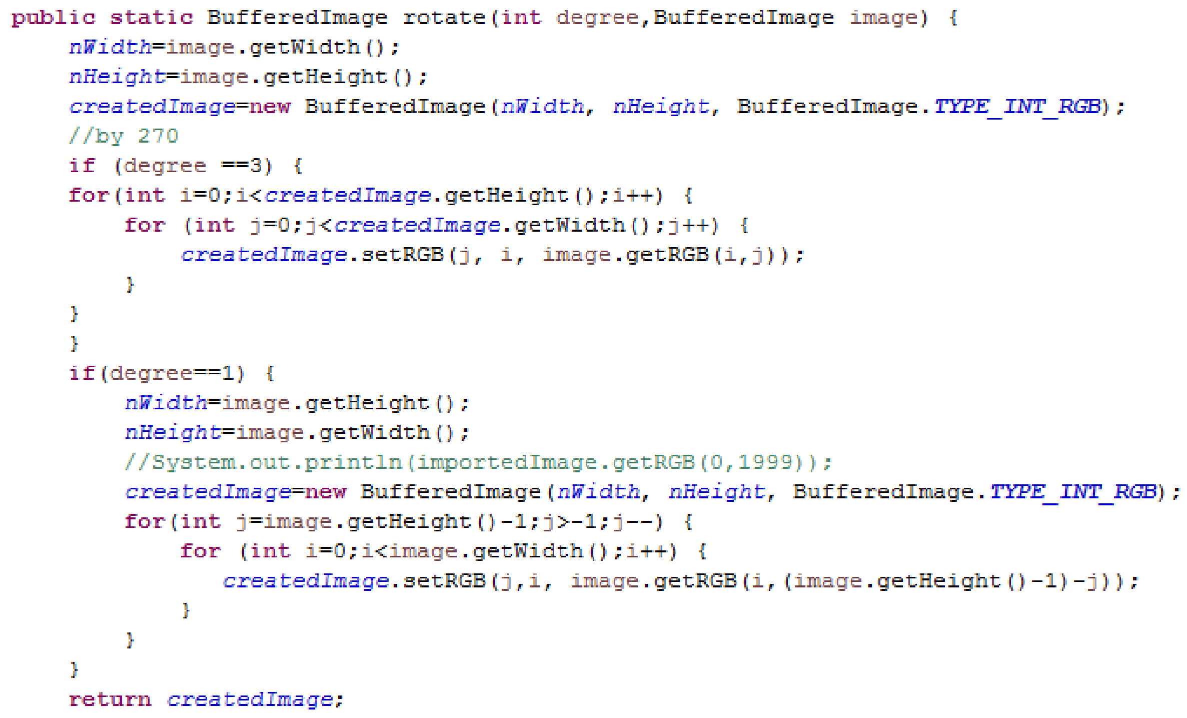
This is getting the vertical mirror image.



```java
public static  BufferedImage flip(BufferedImage image) {
    nWidth=image.getWidth();
    nHeight=image.getHeight();
    createdImage=new BufferedImage(nWidth, nHeight, BufferedImage.TYPE_INT_RGB);
    for(int i=0;i<createdImage.getWidth();i++) {
        for (int j=0;j<createdImage.getHeight()-1;j++) {
            createdImage.setRGB(i,j, image.getRGB(image.getWidth()-1-i,j));
        }
    }
    return createdImage;
}
```

Image rotation

The image can be rotated either in clockwise or anti-clockwise.



```java
public static BufferedImage rotate(int degree,BufferedImage image) {
    nWidth=image.getWidth();
    nHeight=image.getHeight();
    createdImage=new BufferedImage(nWidth, nHeight, BufferedImage.TYPE_INT_RGB);
    //by 270
    if (degree ==3) {
    for(int i=0;i<createdImage.getHeight();i++) {
        for (int j=0;j<createdImage.getWidth();j++) {
            createdImage.setRGB(j, i, image.getRGB(i,j));
        }
    }
    }
    if(degree==1) {
        nWidth=image.getHeight();
        nHeight=image.getWidth();
        //System.out.println(importedImage.getRGB(0,1999));
        createdImage=new BufferedImage(nWidth, nHeight, BufferedImage.TYPE_INT_RGB);
        for(int j=image.getHeight()-1;j>-1;j--) {
            for (int i=0;i<image.getWidth();i++) {
               createdImage.setRGB(j,i, image.getRGB(i,(image.getHeight()-1)-j));
            }
        }
    }
    return createdImage;
```

Grayscaling

The greyscaled image of a the image can be taken by setting the each pixel RGB value to the average of the R,G and B values

```
public static BufferedImage grayscaling(BufferedImage image) {
    BufferedImage createdImage = image;
    int length = image.getWidth();
    int height = image.getHeight();

    for (int i = 0; i < length; i++) {
        for (int j = 0; j < height; j++) {
            int rgb = image.getRGB(i, j);

            int A = (rgb >> 24) & 0xFF;
            int R = (rgb >> 16) & 0xFF;
            int G = (rgb >> 8) & 0xFF;
            int B = (rgb & 0xFF);
            int gray = (R + G + B) / 3;
            rgb = (A << 24) | (gray << 16) | (gray << 8) | gray;
            image.setRGB(i, j, rgb);
        }
    }
   return createdImage;
}
```

Negative

Negative image can be taken by getting the negative value of the each pixel in the image. The negative value of a pixel can be taken by reducing the current RGB value from 255.

```
public static BufferedImage negative(BufferedImage image) {
        BufferedImage createdImage= image;
        int length = image.getWidth();
        int height = image.getHeight();

        for (int i = 0; i < length; i++) {
            for (int j = 0; j < height; j++) {
                Color color = new Color(image.getRGB(i, j));
                int R;
                int G;
                int B;
                R = checkRange(255 - color.getRed());
                G = checkRange(255 - color.getGreen());
                B = checkRange(255 - color.getBlue());
                color = new Color(R, G, B);
                image.setRGB(i, j, color.getRGB());
            }
        }
        return createdImage;
    }
```

Edge Detection

A sudden change of discontinuities of an image is identifies as an edge. Different masks are used in difference algorithms in order to find the edges. Sobel and Robert are one of the mostly using edge detection algorithms in the image processing.

Sobel filter

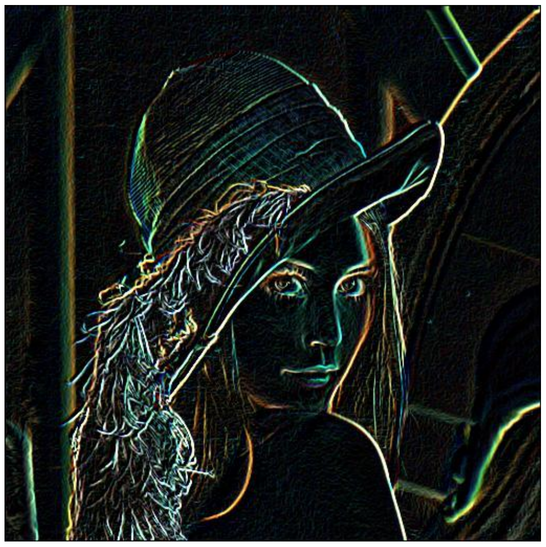
Followings are the horizontal and the vertical mask for the Sobel filter.

Horizontal

1	2	1
0	0	0
-1	-2	-1

Vertical

1	0	-1
2	0	-2
1	0	-1



Robert filter

Followings are the horizontal and the vertical mask for the Sobel filter.

Horizontal

1	2	1
0	0	0
-1	-2	-1

Vertical

1	0	-1
2	0	-2
1	0	-1