Programming Assignment 2

2

CS 474

Steven Hernandez and Adam Lychuk

Date due: 10/28/2020

Date handed in: 10/28/2020

Division of work:

Adam Lychuk implemented problems 1 and 5.

Adam Lychuk wrote the theoretical, implementation, results and discussion of problems 1 and 5.

Steven Hernandez implemented problems 2,3 and 4.

Steven Hernandez wrote the theoretical, implementation, results and discussion of problems 2,3 and 4.

Theory

Part 1

Correlation is a linear operator that computes a weighted sum of integers in a specified region. This technique is often used in the context of images to create an entirely new image as defined in the following formula.

$$g(i,j) = w(i,j) \bullet f(i,j) = \sum_{s=-K/2}^{K/2} \sum_{t=-K/2}^{K/2} w(s,t) f(i+s, j+t)$$

Here g(i,j) is the 2d output image, w(i,j) the weights, and f(i,j) the input image. For each pixel defined by i,j in the input image a new value is computed through correlation by multiplying neighboring pixel values in f with the weights defined in w. The offset for neighboring pixels is defined by s,t. The full sum of the original pixel and its neighbors specified by the mask w is calculated for each pixel in f to create the full output image g. This technique is very versatile as variable weights can provide very different results. An often used application is template matching which measures the similarity between images.

Part 2

Averaging is a linear filter that uses a mask of a given size in order to determine the degree of smoothing on an image. In order to accomplish this, it uses a mask where the values are all set to 1 so when it is multiplied by the value of the corresponding neighborhood of the image, it won't change the values from the image. Depending on the size of the mask, it will increase the degree of smoothing. Once the values are found, they are summed up and divided by the number of elements of the mask in order to normalize the value and bring it back to the pixel value range. T

Gaussian is another linear filter that helps smooth the image; however, compared to averaging, gaussian uses an equation to calculate the values of the mask to be used and helps focus more on the edges of an image. The 2D gaussian equation gives a mask with bigger weights the closer it is to the center and so the values that are further away will be closer or equal to 1. One of the factors that helps determine the size of the mask is also σ which is part of the equation, hence it will control the amount of smoothing. The rest of the process is the same as averaging where the values are added up and then normalized.

$$G_{\sigma}(x,y) = \frac{1}{2\pi\sigma^{2}} exp^{-\frac{x^{2}+y^{2}}{2\sigma^{2}}}$$

Part 3

Median filtering is another smoothing method, but it uses a non-linear operation that is effective for removing noise like "salt and pepper" that can be found in images. This method uses a neighborhood of a certain size, i.e. 7x7, to capture the values of that area and then sort them. Once sorted, the median is chosen to replace all of the values within that neighborhood.

Part 4

Two common sharpening methods are the unsharp masking and the high boosting filtering which can help sharpen an image. Unsharp masking makes this happen by subtracting a low pass image, which could have used a filter such as averaging, gaussian or median, from the original image. This will end up giving a high pass image in return that will have some contrast enhancement.

$$\text{Highpass} = \text{Original} - \text{Lowpass}$$

High boost filtering, another sharpening method, emphasizes sharpening the edges of an image, but may lose details. To achieve this, it uses the original image and multiplies it by a constant which is then added in a high pass image. Depending on the value of the constant, it may end up resulting in an unsharp image if it is equal to 1, but if it is a value greater than 1, then part of the original image is added back.

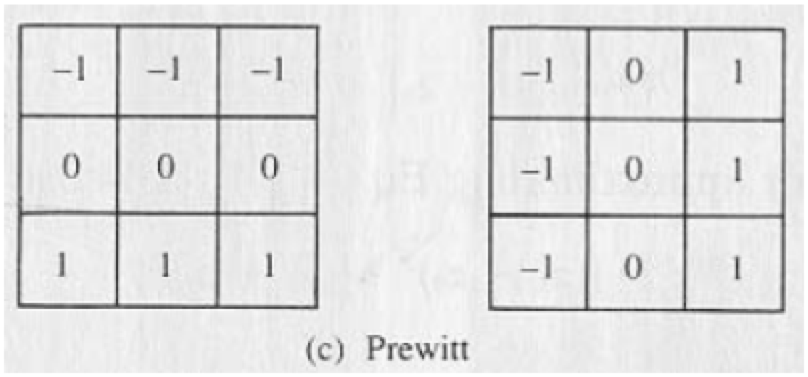
$$\text{High Boost} = (A-1)\ \text{Original} + \text{Highpass}$$

Part 5

The gradient of an image is an often used sharpening method in image processing. The gradient is defined as follows where:

$$grad(f) = \begin{pmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{pmatrix}$$

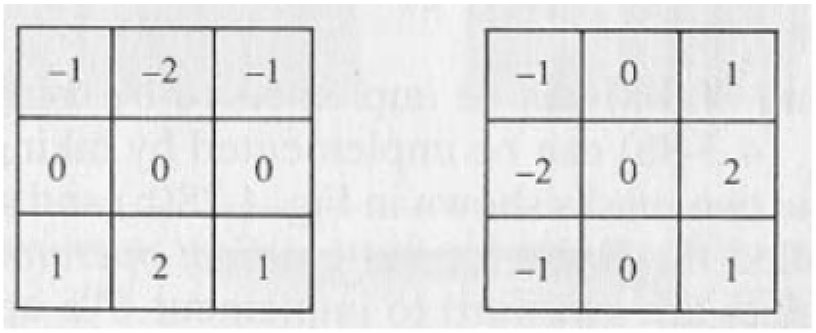
The gradient of f the image is a vector of the partial derivatives of f with respect to x and y. As the result is a derivative and vector, the gradient of an image has both a magnitude and direction. When sharpening an image we amplify edges and the gradient's magnitude defines edge strength and the gradient's direction is perpendicular to the direction of the edge. To implement the gradient over the entire image would be extremely costly in most cases so we instead approximate the partial derivatives by convolving or correlating a mask for the partial of x and the partial of y as defined in part 1. Two popular masks for this purpose are the prewitt and sobel defined as follows:



−1	−1	−1
0	0	0
1	1	1

−1	0	1
−1	0	1
−1	0	1

(c) Prewitt



−1	−2	−1
0	0	0
1	2	1

−1	0	1
−2	0	2
−1	0	1

On the left is the mask defining the partial y and on the right is the mask defining the partial x. The resulting outputs can then be combined using the following formula:

$$\sqrt{\frac{\partial f}{\partial x}^2 + \frac{\partial f}{\partial y}^2}$$

Which results in the gradient magnitude at each pixel. Edges can be found at local maxima and local minima. However because this only provides an increased slope at edges ie. local maxima, it is hard to localize exactly where those edges are along that slope for complex objects. Then the laplacian, a second order derivative, can sometimes provide better results. The laplacian defines edges at "zero-crossings" which end up being at the beginning and end of the slope defined by the first order derivatives. The "zero-crossing" in practice is where the sign of the laplacian flips, and the laplacian is approximated using a single mask that combines the partial x and partial y:

$$\begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline 1 & -2 & 1 \\ \hline 0 & 0 & 0 \\ \hline\end{array} + \begin{array}{|c|c|c|}\hline 0 & 1 & 0 \\ \hline 0 & -2 & 0 \\ \hline 0 & 1 & 0 \\ \hline\end{array} = \begin{array}{|c|c|c|}\hline 0 & 1 & 0 \\ \hline 1 & -4 & 1 \\ \hline 0 & 1 & 0 \\ \hline\end{array}$$

Because the laplacian only uses one mask it is less computationally expensive then prewitt or sobel approximations, but also it only provides gradient magnitude. Gradient direction is lost. Also the laplacian is more sensitive to noise then its counterpart. Both are situationally useful.

Implementation

Part 1

The code to correlate both a variable image and variable mask is contained in the correlate() function. The mask is created through the ImageToMask() function, which simply takes the pixel values of an image and transfers them to a given float array. We iterate through the given image and at each pixel value we correlate the original pixel value with its neighboring pixels weighted by the given variable mask. For each neighbor we calculate an offset describing which mask value should multiply which neighbor. If the neighbor is out of the bounds we wrap around the image to provide padding. After all weighted values have been summed for the mask we output the pixel value to the output image. After receiving the output image the values are then normalized using min/max normalization to 0-255. This is done in the Normalize255() function. The max and min values of the pixels in the image are found using a simple loop and then we loop through the image again normalizing the values using that max and min.

Part 2

This part of the assignment needed to read an image and implement the averaging and gaussian methods along with using masks of sizes 7x7 and 15x15. The first step lets the user decide what method and size to use. Once it is decided, there are 'if-statements' to guide the program to its respective block of code where the image will be processed per the method and size requested. The image is read and the values are stored in a vector so the values can be accessed later. Then, a 2Darray is made to fit not only the image values, but a border of zeros so it will surround all the values and make it easier later on to apply the mask. The mask will then go over each value of the image to get its neighboring values. If it is an average filter being used, then it will have the values summed up and normalized. On the other hand, if it is the gaussian method, it will multiply the values with the respective gaussian values, summed and normalized. The new values are then placed into a new vector which will be used to set the values to the new image.

Part 3

To better understand how useful the median method is, the assignment asked for images to have "salt and pepper" noise added in. Hence, images with a 30 and 50 percent noise were made. The program does the same process as part two to read in the values in a 2D array with zeros to ease the masking part of the values. The mask in this case took the values within the neighborhood and sorted them from smallest to largest. Once this was done, the median was found within the values and used to replace the values in the neighborhood where the mask was currently over.

Part 4

Unsharp filtering and high boost filtering were implemented in this part. The unsharp filtering had two images read, an original image and an image with the 7x7 mask gaussian filter, to be able to make the new image. The program simply had the smoothed image subtract from the original image which then had the values stored in a vector where it would later be used for the new image. This resulted in a high pass image being made. The next part is the high boost filter which was implemented in another file. The program first has the user input a constant for A. The program then reads in the original image and the high pass image that was made before. With both images being read, the program then has the constant subtracted by 1 before being multiplied with the values of the original image and then have the values of the high pass image added. The new values are floored and placed into a vector to later have them read into the new image.

Part 5

Using the Correlate() function from part 1 we correlate the prewitt, sobel, and laplacian masks with the sf and lenna images. The gradient magnitude for prewitt and sobel is calculated using the Magnitude() function. We take the partial x and partial y images and then at each pixel value in the gradient image we take the floor of the square root of the square of the partial x and partial y.

Results and Discussion

Part 1

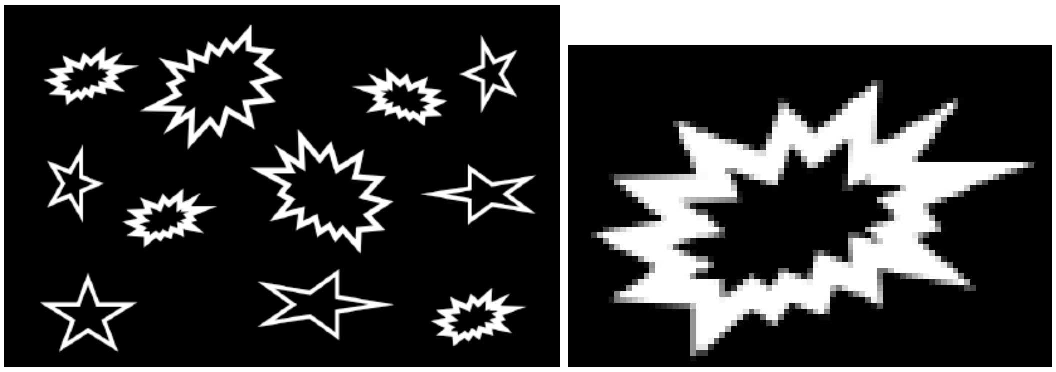


Figure 1: (a) original image (b) correlated image

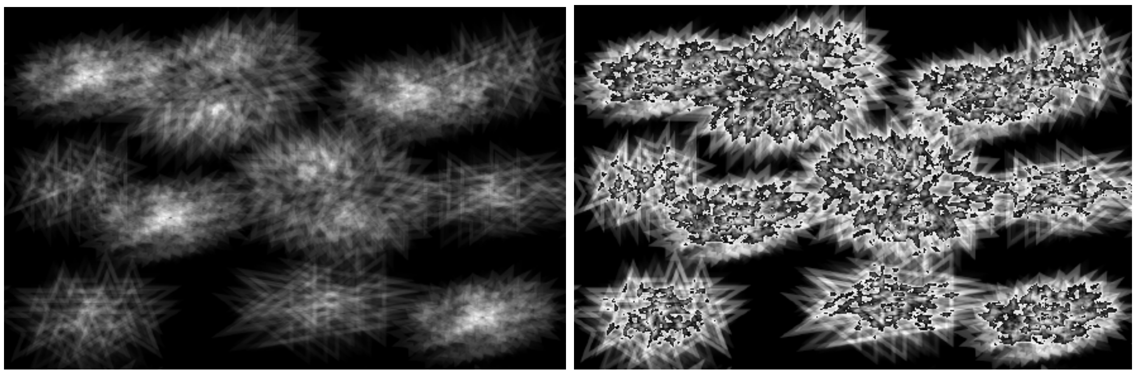


Figure 2: (a) correlated image normalized using min/max normalization from 0-255 (b) correlated image

By correlating the image Figure 1 (a) with the image Figure 1(b) we find local minima and maxima at locations where the pattern might have appeared in the image. Using thresholding we could find the location of objects similar to the given mask, but as we can see in the results in Figure 2 (a)(b) there are limits on accuracy. Objects are correlated that do not match the original mask and orientation needs to be similar for the best results. A more complicated algorithm with a variable mask or a combination of the correlation of the same pattern transformed using an affine transform might find more success for template matching.

Part 2

Original images

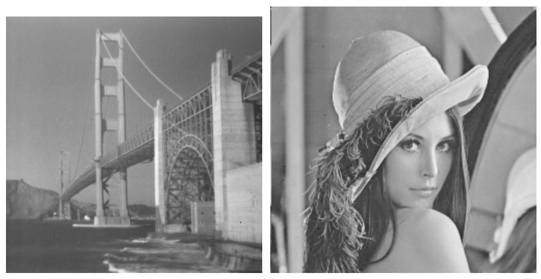


Figure 3: (a) sf.pgm (b) lenna.pgm

Average

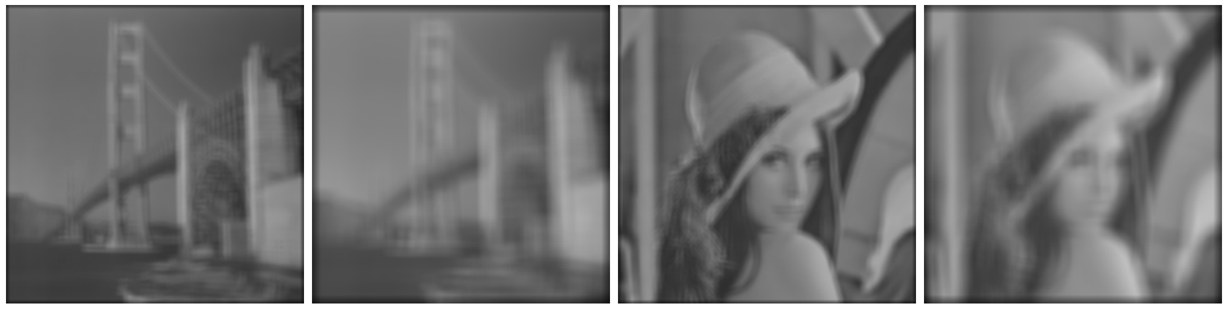


Figure 4: (a) sf with 7x7 average filter (b) sf with 15x15 average filter (c) lenna with 7x7 average filter (d) lenna with 15x15 average filter

Gaussian

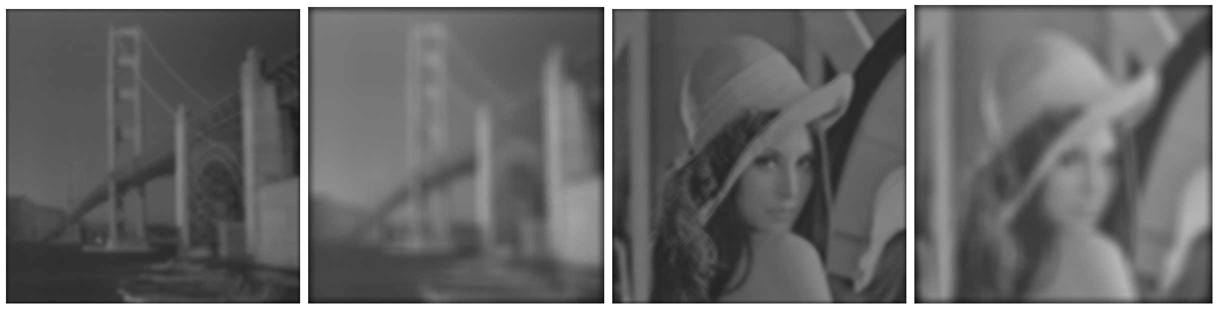


Figure 5: (a) sf with 7x7 gaussian filter (b) sf with 15x15 gaussian filter (c) lenna with 7x7 gaussian filter (d) lenna with 15x15 gaussian filter

Both averaging and gaussian filters are linear smoothing methods that help blur the image and they seem to have done their job pretty well as the images have been blurred. When comparing the two methods however, the gaussian giving more weight to the center of the masks seems to have helped preserve the edges of the figures within the images as seen as above. This is more noticeable in the 15x15 masks of both the sf and lenna images.

Part 3

Salted images

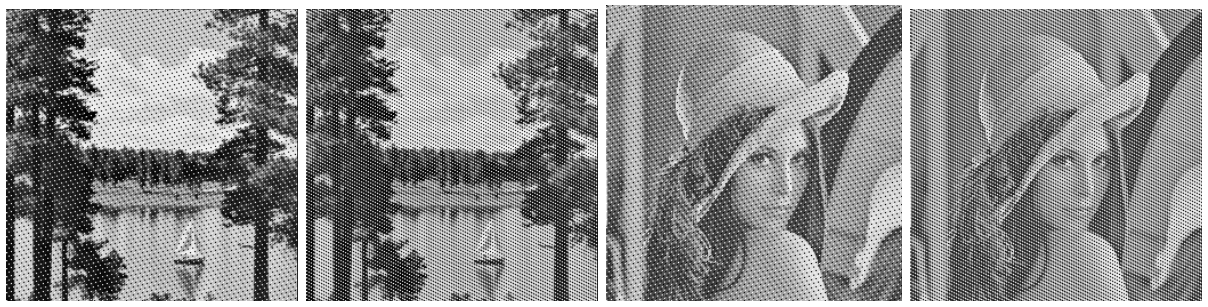


Figure 6: (a) boat with 30% noise (b) boat with 50% noise (c) lenna with 30% noise (d) lenna with 50% noise

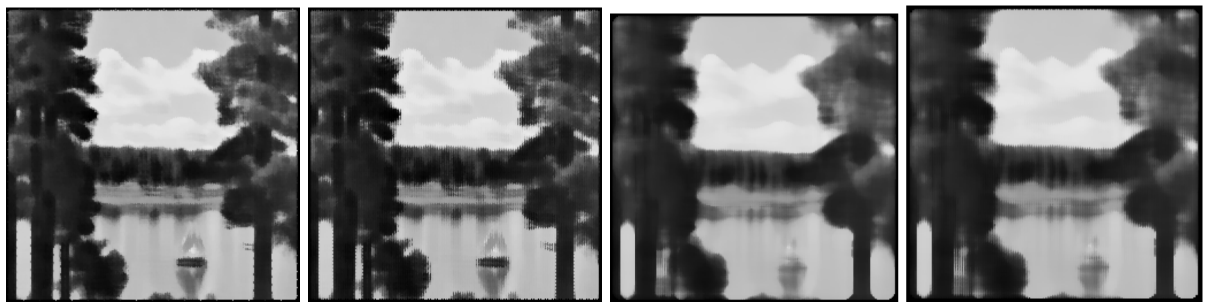


Figure 7: (a) boat with 30% noise and 7x7 median filter (b) boat with 50% noise and 7x7 median filter (c) boat with 30% noise and 15x15 median filter (d) boat with 50% noise and 15x15 median filter

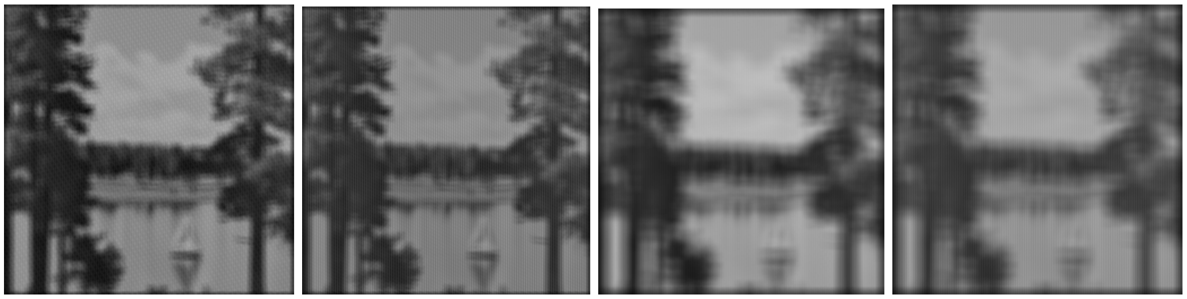


Figure 8: (a) boat with 30% noise and 7x7 average filter (b) boat with 50% noise and 7x7 average filter (c) boat with 30% noise and 15x15 average filter (d) boat with 50% noise and 15x15 average filter

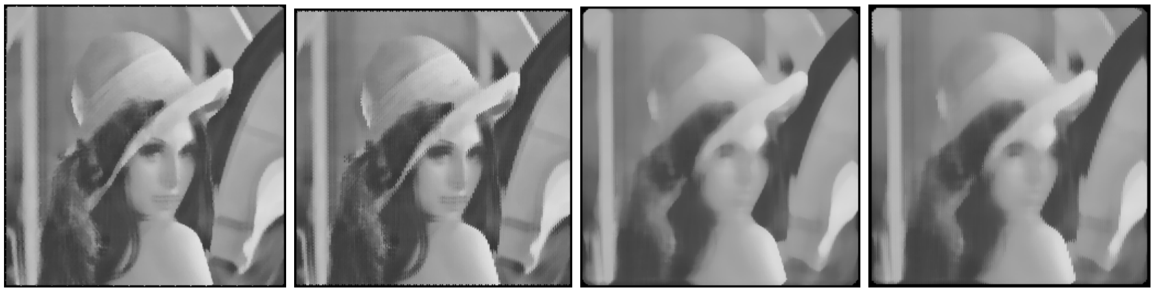


Figure 9: (a) lenna with 30% noise and 7x7 median filter (b) lenna with 50% noise and 7x7 median filter (c) lenna with 30% noise and 15x15 median filter (d) lenna with 50% noise and 15x15 median filter

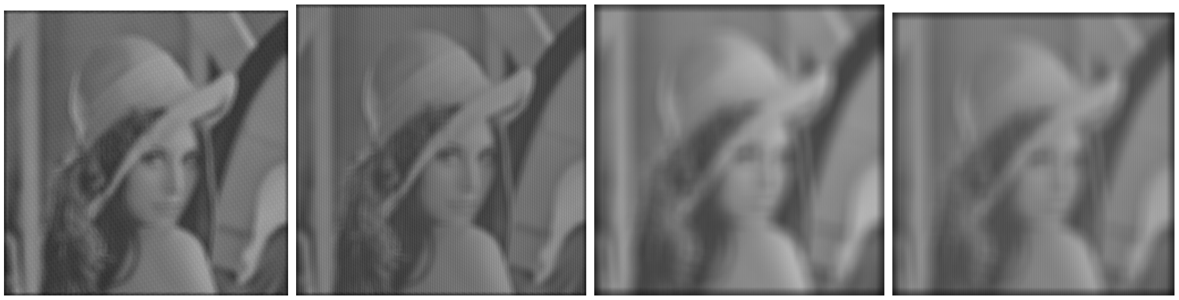


Figure 10: (a) lenna with 30% noise and 7x7 average filter (b) lenna with 50% noise and 7x7 average filter (c) lenna with 30% noise and 15x15 average filter (d) lenna with 50% noise and 15x15 average filter

This part of the assignment asked for the lenna and boat images to be corrupted with noise with about 30% and 50% “salt and pepper” noise added to each in order to be used with the median filter. In order to compare the results of the median filter, the corrupted images were passed through the averaging filter as well. The results show that both seem to have done a good job in removing the “salt and pepper” noise added to the images; however, the averaging images were smoothed a lot more compared to the median images. This is very obvious when comparing the images that used the 15x15 masks for both the 30% and 50% corrupted images.

Part 4

7x7 gaussian mask

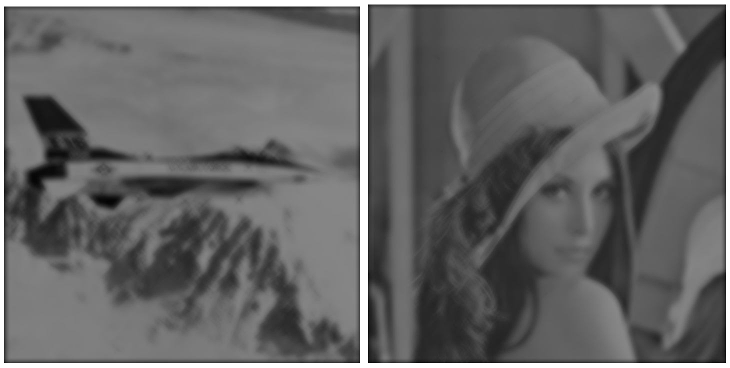


Figure 11: (a) f_16.pgm with 7x7 gaussian filter (b) lenna.pgm with 7x7 gaussian filter

Unsharp

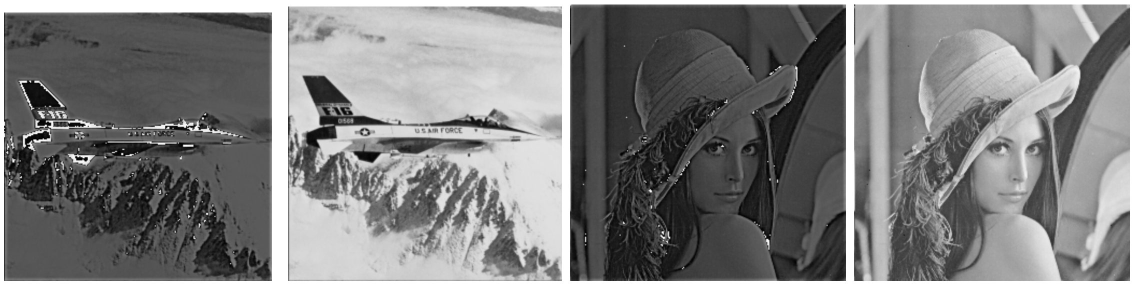


Figure 12: (a) f_16.pgm with unsharp filter (b) original f_16 (c) lenna.pgm with unsharp filter (d) original lenna

High Boost

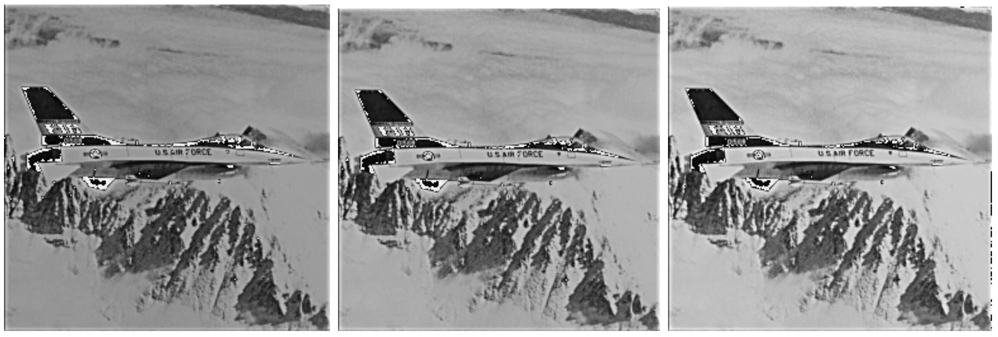


Figure 13: (a) high boost and A = 1.3 (b) high boost and A = 1.4 (c) high boost and A = 1.5

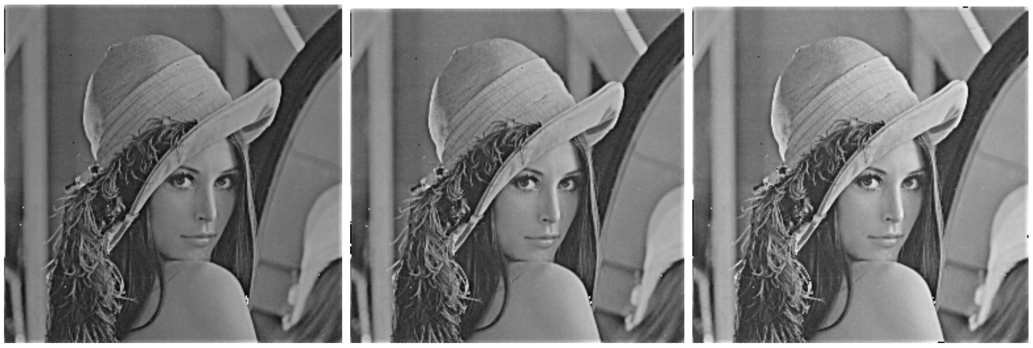


Figure 14:(a) high boost and A = 1.3 (b) high boost and A = 1.4 (c) high boost and A = 1.5

In order to get a high pass image, a smoothed (lowpass) image was to be subtracted from the original version. The assignment asked for the smoothed images to have used the 7x7 gaussian mask for the lowpass image. After having passed the images through the unsharp filter, the images seem to have been sharpened a bit more compared to the original. It is a bit harder to see it in the f_16 unsharp image, but it is more clear when comparing the original lenna and the unsharpened lenna images. As for the high boost images, there was detail lost along the way, this can especially be seen in the f_16 unsharp images as well as the lenna images where even pixel values seem to have been lost. Nevertheless, the method did its job as the edges are more refined as seen in the f_16 images where the clouds edges can be seen more clearly and Lenna's hat seems a bit more sharpened. Different 'A' values were used , but using a value of 1.4 seems to have done a better job of sharpening the edges and keeping the contrast of the image closer to that of the original.

Part 5

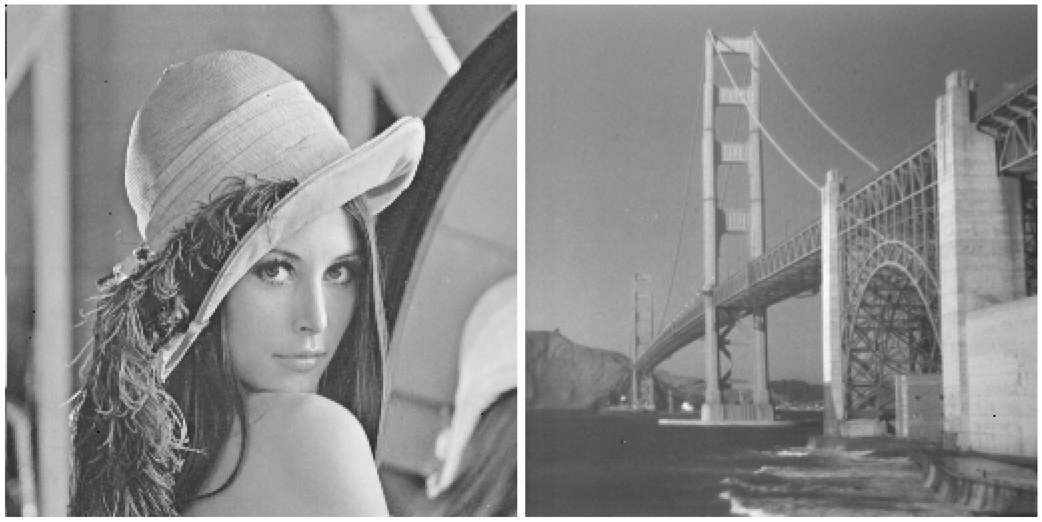


Figure 15: (a) Original lenna (b) Original sf

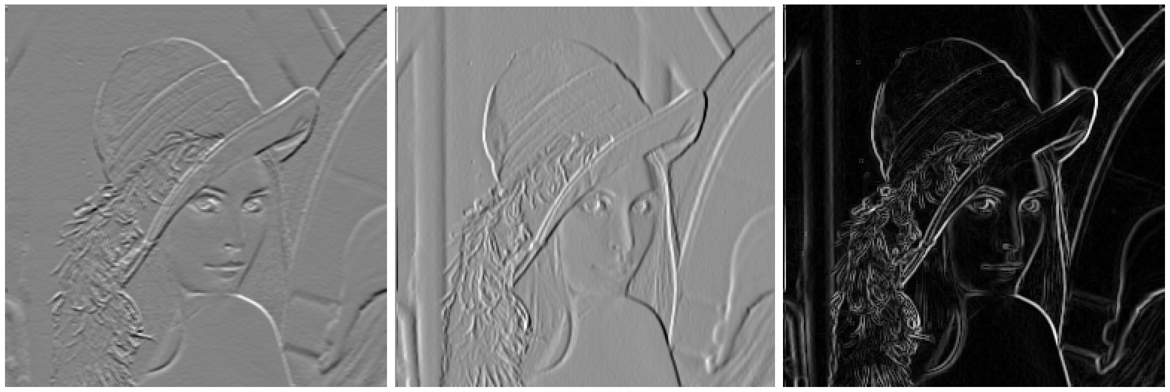


Figure 16: (a) Prewitt lenna partial y (b) Prewitt lenna partial x (c) Prewitt lenna gradient magnitude

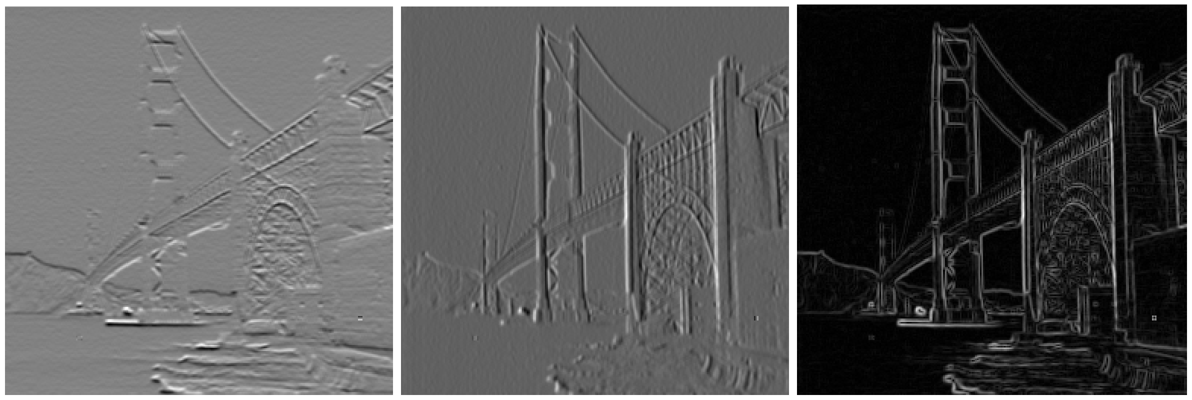


Figure 17: (a) Prewitt sf partial y (b) Prewitt sf partial x (c) Prewitt sf gradient magnitude

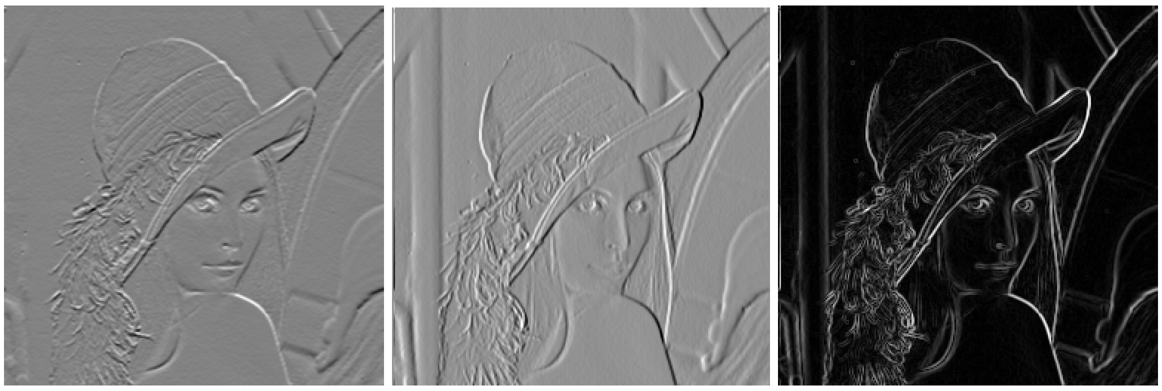


Figure 18: (a) Sobel lenna partial y (b) Sobel lenna partial x (c) Sobel lenna gradient magnitude

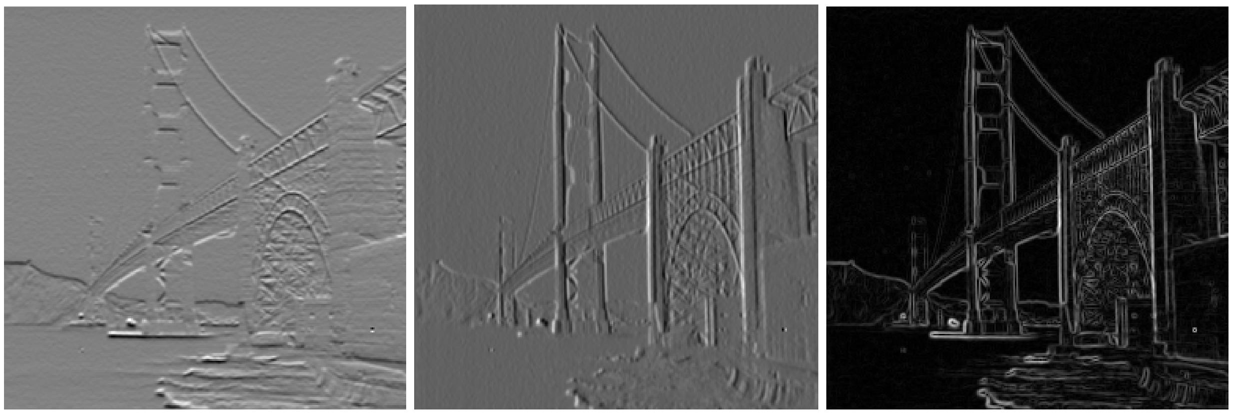


Figure 19: (a) Sobel sf partial y (b) Sobel sf partial x (c) Sobel sf gradient magnitude

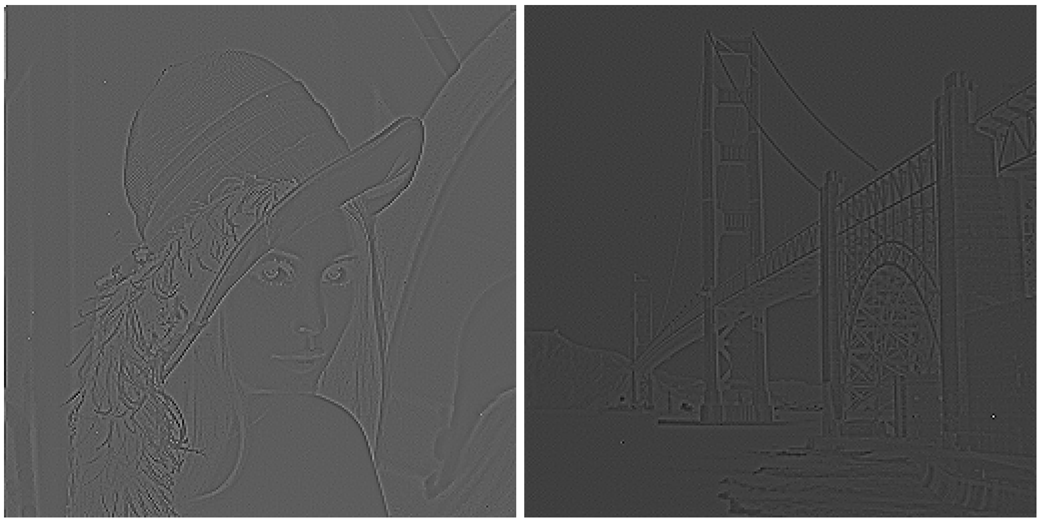


Figure 20: (a) Laplace lenna (b) Laplace sf

All the above algorithms define the edges in the image. In figure 16 and 17 the prewitt masks define the edges well when there are defined lines but struggle with the complexity of the bridges trusses and Lenna's hair. Edges are defined within these complex objects muddying the result. The gradient magnitude that combines the information from the partial x and partial y provides the cleanest result. In figure 18 and 19 the Sobel mask provides similar results with an increase in definition. Intuitively this is because of the -2 and 2 which increase slope definition at its peak.

This in turn increases the gradient magnitude at edges. The laplacian in figure 20 provides the most accurate results. Hair in the lenna image is still problematic, but the trusses in the sf image are well defined. With the computational efficiency and lack of noise in both images the laplacian seems like the best choice for edge sharpening as long as we do not care about the gradient direction information.

Source Code

Part 1

```
void Correlate(ImageType &input, ImageType &output, Point size,
            float **kernel, Point pad, int boundaries)
{
  int rows, cols, levels, dummy;
  input.getImageInfo(rows, cols, levels);
  for (int i = 0; i < rows; ++i)
  {
    for (int j = 0; j < cols; ++j)
    {
      float sum = 0;
      Point sample;
      for (int k = 0; k < size.x; ++k)
      {
        for (int l = 0; l < size.y; ++l)
        {
          sample.x = abs(i + k - pad.x);
          sample.y = abs(j + l - pad.y);
          if (sample.x >= rows)
            sample.x = rows - 1 - sample.x % rows;
          if (sample.y >= cols)
            sample.y = cols - 1 - sample.y % cols;
          sum += static_cast<float>(input.getPixelVal(sample.x, sample.y,
dummy)) * kernel[k][l];
```

```
          }
        }
        output.setPixelVal(i, j, sum);
      }
    }
  }
  void Normalize255(ImageType& image) {
    int rows, cols, levels, val, newval;
    int max = 0;
    int min = 0;
    image.getImageInfo(rows, cols, levels);
    for (int i = 0; i < rows; ++i)
    {
      for (int j = 0; j < cols; ++j)
      {
        if (image.getPixelVal(i,j,val) > max) {
          max = val;
        }
        else if (image.getPixelVal(i,j,val) < min) {
          min = val;
        }
      }
    }
    int range = max - min;
    int newmin = 0;
    int newmax = 255;
    int newrange = newmax - newmin;
    cout << range;
    for (int i = 0; i < rows; ++i) {
      for (int j = 0; j < cols; ++j) {
        newval = newmin + (((image.getPixelVal(i,j,val) - min) * newrange) /
range);
        image.setPixelVal(i,j,newval);
      }
    }
```

```
}
void ImageToMask(ImageType image, float **mask)
{
  int rows, cols, levels, dummy;

  image.getImageInfo(rows, cols, levels);
  for (int i = 0; i < rows; ++i)
  {
    for (int j = 0; j < cols; ++j)
    {
      mask[i][j] = 1.0f / ((static_cast<float>(image.getPixelVal(i, j, dummy)) *
static_cast<float>(rows * cols))) * 1000;
      if (image.getPixelVal(i, j, dummy) == 0) {
        mask[i][j] = 0;
      }
    }
  }
}
```

Part 2

```
if(filter == 'a' && size == 7)
{
  int BufferArray7[262][262];
  for(int i = 0; i < 262;i++)
  {
    for(int j = 0; j < 262; j++)
    {
      BufferArray7[i][j] = 0;
    }
  }

  int counterj = 0;
  //Fill in buffer array with the image values so they are surrounded by zeros
  for(int i = 0; i < 262; i++)
  {
    for(int j = 0; j < 262; j++)
    {
      if((i > 2) && (i < 259))
      {
```

```
      if((j > 2) && (j < 259))
      {
        BufferArray7[i][j] = TempImArr[counterj];
        counterj++;
      }
    }
    else
    {
      BufferArray7[i][j] = 0;
    }
  }
}

int MaskArray[7][7];
int sum = 0;

for(int i = 3; i < 259; i++)
{
  for(int j = 3; j < 259; j++)
  {
    //This next nested loops are to go through the mask
    //neighborhood of the current index
    for(int a = 0; a < 7; a++)
    {
      int differenceA;
      int Aflag = 1; // 0 = less, 1 = equal, 2 = more
      if(a < 3)
      {
        differenceA = 3 - a;
        Aflag = 0;
      } else if(a == 3)
      {
        differenceA = 0;
        Aflag = 1;
      }else if(a > 3)
      {
       differenceA = a - 3;
       Aflag = 2;
      }

      for(int b = 0; b < 7; b++)
      {
        int differenceB;
        int Bflag = 1; // 0 = less, 1 = equal, 2 = more
        if(b < 3)
        {
          differenceB = 3 - b;
```

```
            Bflag = 0;
          } else if(b == 3)
          {
            differenceB = 0;
            Bflag = 1;
          }else if(b > 3)
          {
           differenceB = b - 3;
           Bflag = 2;
          }

         //Once flags are set, the index will be calculated to find correct
         //index of the mask array

         if((Aflag == 0 && Bflag == 0) ||(Aflag == 1 && Bflag == 1))
         {
           MaskArray[a][b] = BufferArray7[i-differenceA][j-differenceB];
         }
         else if(Aflag == 0 && Bflag == 2)
         {
           MaskArray[a][b] = BufferArray7[i-differenceA][j+differenceB];
         }
         else if(Aflag == 2 && Bflag == 0)
         {
           MaskArray[a][b] = BufferArray7[i+differenceA][j-differenceB];
         }
         else if(Aflag == 2 && Bflag == 2)
         {
           MaskArray[a][b] = BufferArray7[i+differenceA][j+differenceB];
         }
       }
     }

     //For averaging, it sums up the mask's values
     for(int a = 0; a < 7; a++)
     {
       for(int b = 0; b < 7; b++)
       {sum += MaskArray[a][b];}
     }

     //Normalize the values
     sum = sum/49;
     //new values pushed into vector
     NewValues.push_back(sum);
     sum = 0;
   }
 }
```

```
}
```

Part 3

```
//Same process to find neighborhood with
      //mask like averaging and gaussian

      //An array gets made to be used to sort the
      //values of the neighborhood
      int sortArray[49];
      int sortIndex = 0;

      //transfers values captured from mask
      //to the 1D array
      for(int a = 0; a < 7; a++)
      {
        for(int b = 0; b < 7; b++)
        {
          sortArray[sortIndex] = MaskArray[a][b];
          sortIndex++;
        }
      }
      //Sorts the values of the neighborhood from
      //smallest to largest
      int asize = sizeof(sortArray) / sizeof(sortArray[0]);
      sort(sortArray, sortArray + asize);

      // Chooses the median after being sorted
      sum = sortArray[24];
      //Median value gets pushed to vector
      NewValues.push_back(sum);
      sum = 0;
```

Part 4

```
vector <int> NewValues;
//////////////////// unsharp ///////////////////////////////////////////////////
for(int i = 0; i< N; i++)
{
  for(int j = 0; j < M; j++)
  {
    //variable to get new value
    int val3;
    image.getPixelVal(i,j,val);
    image2.getPixelVal(i,j,val2);
    //variable equals the value of the smoothed
    //image minus the value of original image
    val3 = val - val2;
    //new value gets pushed into vector
    NewValues.push_back(val3);
  }
}

int VecIndex = 0;
for(int i = 0; i < N; i++)
{
  for(int j = 0; j < M; j++)
  {
    //call getPixelVal just to get 'val'
    image.getPixelVal(i,j,val);
    //replace original values with new equalized values and DONE!
    image.setPixelVal(i,j,NewValues[VecIndex]);
    VecIndex++;
  }
}
```

```
//////////////////// high boost ////////////////////////////////////////////////
for(int i = 0; i< N; i++)
{
  for(int j = 0; j < M; j++)
  {
    //variable to hold new value
    float val3;
    image.getPixelVal(i,j,val);
    image2.getPixelVal(i,j,val2);
    //Constant - 1 get multiplied by the original image values
    //and then gets added the highpass image value
    val3 = ((constant-1) * ((float)val)) + ((float)val2);
    //value gets rounded down
    floor(val3);
    //makes sure value is of type int
    val3 = (int)val3;
    //pushes new value into vector
    NewValues.push_back(val3);
  }
}

int VecIndex = 0;
for(int i = 0; i < N; i++)
{
  for(int j = 0; j < M; j++)
  {
    //call getPixelVal just to get 'val'
    image.getPixelVal(i,j,val);
    //replace original values with new equalized values and DONE!
    image.setPixelVal(i,j,NewValues[VecIndex]);
    VecIndex++;
  }
}
```

Part 5

```
void Magnitude(ImageType& out, ImageType& part_x, ImageType& part_y)
{
  int rows, cols, levels, val, newval, part_X, part_Y;
  int max = 0;
  int min = 0;
  out.getImageInfo(rows, cols, levels);
  for (int i = 0; i < rows; ++i)
  {
    for (int j = 0; j < cols; ++j)
    {
      part_x.getPixelVal(i,j,part_X);
      part_y.getPixelVal(i,j,part_Y);
      newval = floor(sqrt((part_X*part_X) + (part_Y*part_Y)));
      out.setPixelVal(i,j,newval);
    }
  }
}
```