5.2.3 SPATIAL FILTERS.

Formula:

$$g(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(x, y) f(x + s, y + t)$$

Where:
$g(x, y)$ is the final image.
$w(s, t)$ is the filter or mask.
$f(x + s, y + t)$ is the pixel value of the input related to a postion on the mask or filter

(Gonzalez and Woods, 2002, p. 177)

example filter relative to x, y

```
w(-1,-1), w(-1,0), w(-1,-1)
w(0,-1),  w(0,0),  w(0,1)
w(1,-1),  w(1,0),  w(1,1)
```

Figure 23: Neighborhood offset coordinates

(Gonzalez and Woods, 2002, p. 177)

However as Cocoa on OS X uses the Cartesian coordinate system (Cocoa drawing Guide, 2012) this would be:

```
w(-1,1),  w(0,1),   w(1,1)
w(-1,0),  w(0,0),   w(1,0)
w(-1,-1), w(0,-1),  w(1,-1)
```

Figure 24: Neighborhood offset Cartesian coordinates

So the first iteration of the inner loop at f x = 5, y = 5

```
s = -1
t = -1

g(x, y) += f(x = 4, y = 4) * w(-1, -1)
```

Figure 25: Spatial Filter iteration 1

The second iteration at f x = 5, y = 5

```
s = -1
t = 0

g(x, y) += f(x = 4, y = 5) * w(-1, 0)
```

Figure 26: Spatial Filter iteration 2

5.2.3.1 SMOOTHING FILTERS.

When applying a filter to an image there needs to be consideration for edge pixels as the filter neighborhood is centred around each pixel with the neighborhood overlapping the edge of the image when computing the edge pixels.

Two options to consider are:

1. Padding the image by the length of the new filter to the edge of the filter or $\left|\frac{sizeoffilter}{2}\right|$ requiring the representation to be computed for each size of filter before processing the filter.
2. Pad the loop over the image by abs(size of the filter / 2) leaving the image only partially filtered and create artefacts around the edges of the image.

5.2.3.1.1 SIMPLE AVERAGING FILTER.

For a simple averaging filter each value of the filter is the same. Simply computed as $\frac{1}{sizeofthefilter}$ this value is then applied to the neighbourhood. Dependant on the size of the filter specified the range of `s` and `t` variable will change.

```
w = 1.0 / size

for each pixel x, y

for s = -1 to 1
  for t = -1 to 1

    val += pixel[x + s, y + t] * w

  end
end

pixel[x, y] = val
```

Figure 27: Simple averaging filter pseudo code.

As the use of a simple averaging filter will degrade the edges of pen marks, these will be used sparingly and only of a small filter size.

```
// main.m
NSBitmapImageRep *smoothed = [ip smoothWithSimpleAveragingFilterOfSize:5];


// IP.m
- (NSBitmapImageRep *) smoothWithSimpleAveragingFilterOfSize:(int)size
{

    // create a represenation of the original image
    NSBitmapImageRep *representation = [self
grayScaleRepresentationOfImage:self.image];
    unsigned char *original = [representation bitmapData];

    // create a representation that will store the smoothed image.
    NSBitmapImageRep *output = [self grayScaleRepresentationOfImage:self.image];
    unsigned char *smoothed = [output bitmapData];

    float weight = 1.0 / (float)(size * size); // e.g. 1/(3 * 3) = 0.111
    int padding = (size - 1) / 2.0;  // pad the image

    // iterate over each pixel of the image
    for ( int y = padding; y < self.height - padding; y++ ) {
        for (int x = padding; x < self.width - padding; x++) {

            // find the centre pixel.
            int centre = x + y * self.width;
            int val = 0;

            // iterate over the filter
            for (int s = -padding; s < (padding + 1); s++) {
                for (int t = -padding; t < (padding + 1); t++) {

                    // offset the current x, y
                    int index = (x + s) + ((y + t) * self.width);
                    // add the values
                    val += original[index] * weight;

                }
            }

            // reject values over 255 to prevent
            if ( val > 255 ) val = 255;
            // apply the new value to centre of the filter
            smoothed[centre] = val;
        }
    }

    return output;
}
```

Figure 28: Simple averaging filter implementation.

OUTPUT.

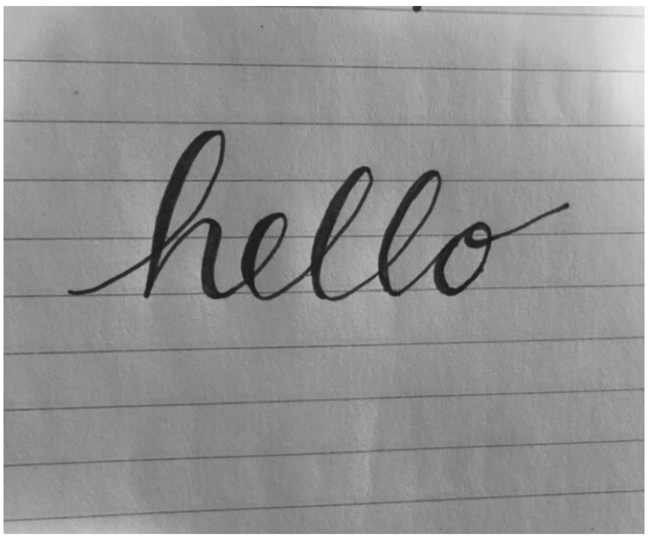

Figure 29: Original Image Grayscaled

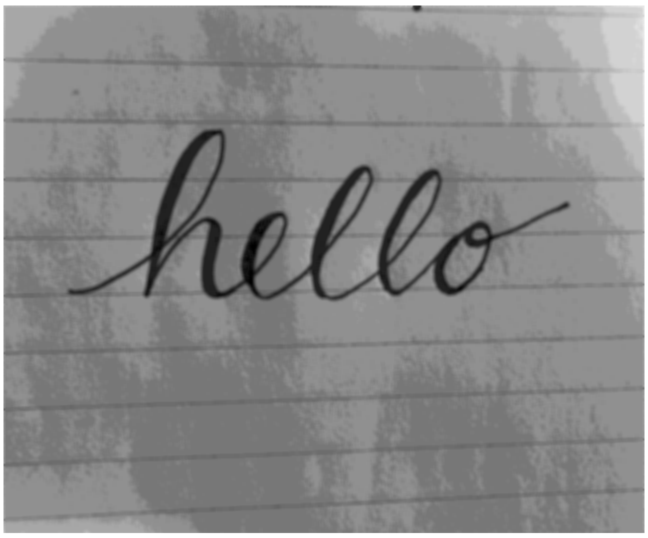

*Figure 30: 5 * 5 Simple Averaging Filter*

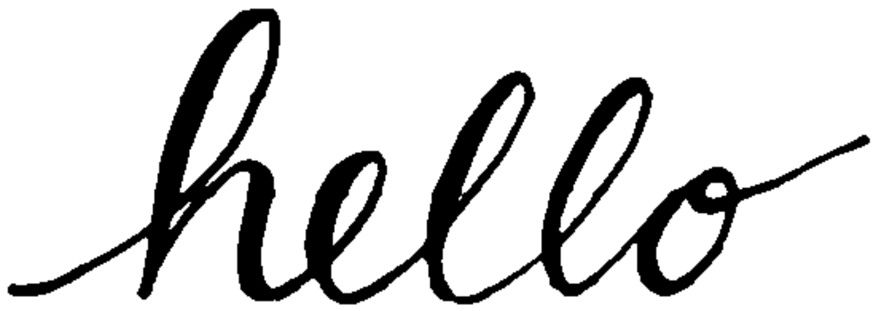

Figure 31: Thresholded 30%

5.2.3.1.2 WEIGHTED AVERAGING FILTER.

The aim of a weighted averaging filter is to reduce the blurring in the noise removal process. With a filter that is more greatly weighted towards the centre and more importance given to the centre pixel (Gonzalez and Woods, 2002, p. 120). Therefore, when centre pixel is part of or the edge of an object of the image this will maintain a higher prominence.

For the filter:

```
1, 2, 1
2, 4, 2 x (1/16)
1, 2, 1
```

Figure 32: Weighted averaging filter (Gonzalez and Woods, 2002, p. 120)

```
filter = [1, 2, 1, 2, 4, 2, 1, 2, 1] * (1 / 16)
i = 0

for each pixel x, y

for s = -1 to 1
  for t = -1 to 1
    val += pixel[x + s, y + t] * filter[i++]
  end
end

pixel[x, y] = val
```

Figure 33: Weighted Averaging filter pseudo code.

```
// As with simpleAveragingFilter

int weights[9] = {1, 2, 1, 2, 4, 2, 1, 2, 1};
float filter[9];

for (int i = 0; i < 9; i++)
{
    filter[i] = (float)weights[i] / 16.0;
}

// As with simpleAveragingFilter

   val += original[index] * filter[i++];

// As with simpleAveragingFilter
```

Figure 34: Weighted Averaging filter implementation.

5.2.3.1.3 MEDIAN FILTER.

The median filter applies the *median* value of the neighborhood to origin pixel.

```
filter[size]
i = 0

for s = -1 to 1
  for t = -1 to 1
    filter[i++] = pixel[x + s, y + t]
  end
end

pixel[x, y] = medianValueFromArray:filter ofSize:size

// medianValueFromArray.

middle = size / 2
sort filter

return filter[middle]
```

Figure 35: Median Filter Pseudo Code

```
// main.m

NSBitmapImageRep *median = [ip reduceNoiseWithMedianFilterOfSize:9];


// IP.m
- (NSBitmapImageRep *) reduceNoiseWithMedianFilterOfSize:(int)size
{
    // create representations.
    // as with smoothWithSimpleAveragingFilterOfSize

    int padding = (size - 1) / 2.0;
    int filter[size * size];

    for ( int y = padding; y < self.height - padding; y++ ) {
        for (int x = padding; x < self.width - padding; x++) {

            int centre = x + y * self.width;
            int i = 0;

            for (int s = -padding; s < (padding + 1); s++) {
                for (int t = -padding; t < (padding + 1); t++) {

                    int index = (x + s) + ((y + t) * self.width);
                    filter[i++] = original[index];

                }
            }

            smoothed[centre] = [self getMedianFromArray:filter ofSize:size];
        }
    }

    return output;
}


- (int) getMedianFromArray:(int [])arr ofSize:(int)size
{
    int middle = (int)(size / 2);
    [self bubbleSort:arr ofSize:size];
    return arr[middle];
}
```

Figure 36: Median Filter Implementation.

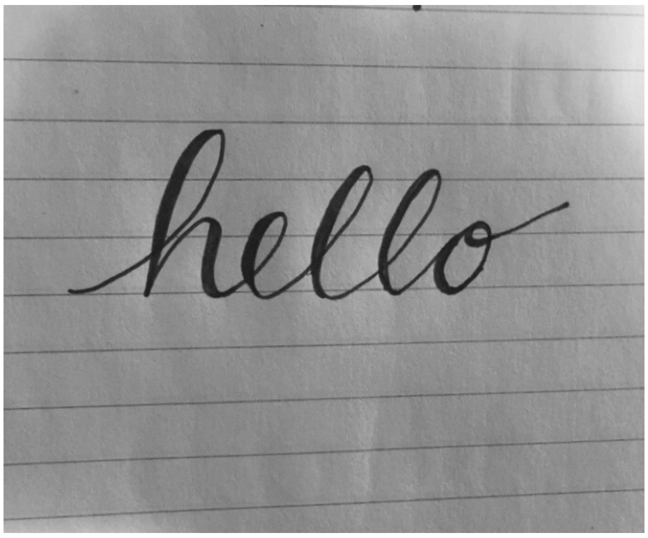

Figure 37: Original Image Grayscaled

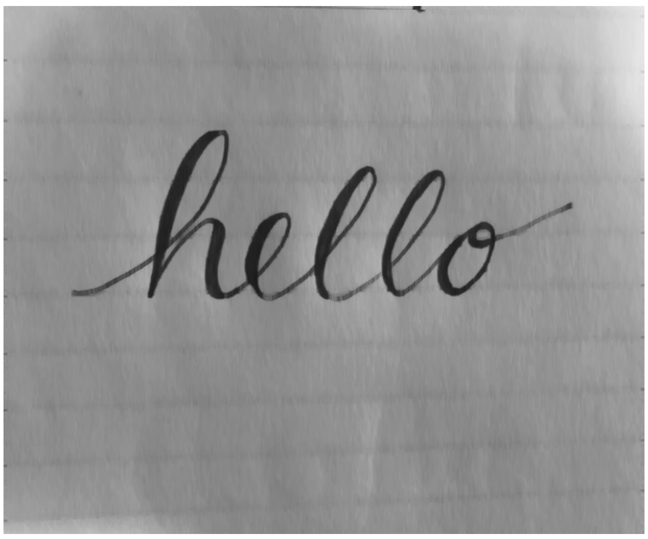

*Figure 38: 9 * 9 Median Filter*

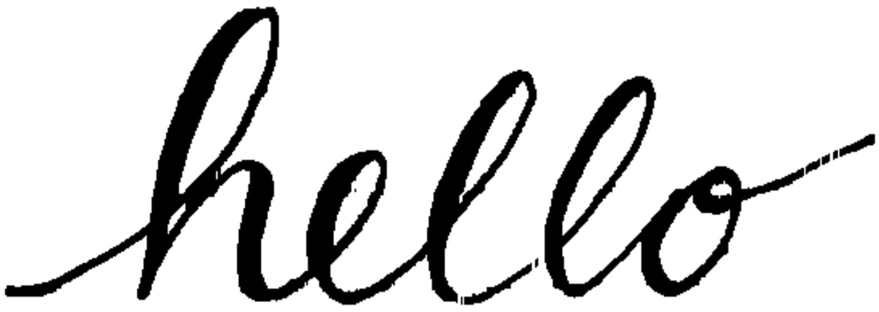

Figure 39: Filtered Image Thresholded 40%

Results of the 9 * 9 Median filter show artefacts created on the upstroke of the “h” and lost information on the exit of the first “l” and on the tail of the “o”

5.2.3.1.4 MIN/MAX FILTER.

The max and min filters differ from the median filter in only one detail, the value set is either the maximum or minimum values of the collected neighborhood rather then the median.

```
// main.m
NSBitmapImageRep *max = [ip reduceNoiseWithMaxFilterOfSize:11];


// as with median filter

- (int) minFromArray:(int [])arr ofSize:(int)size
{
    [self bubbleSort:arr ofSize:size];
    return arr[0];
}
```

Figure 40: Min/max Filter Implementation

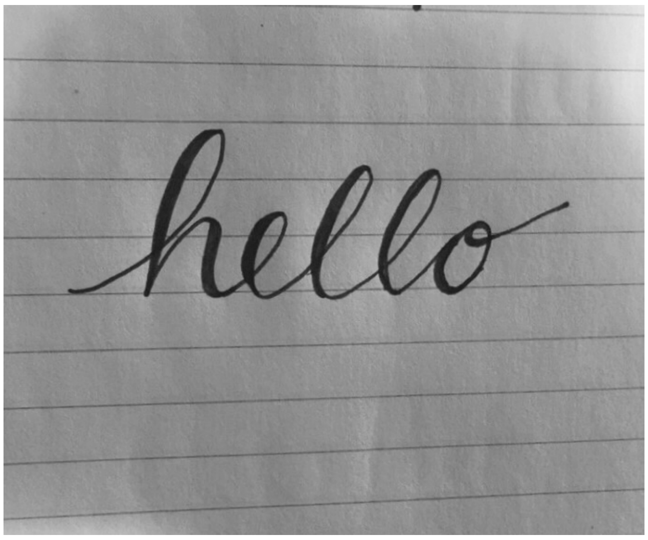

Figure 41: Original Image Gray scaled

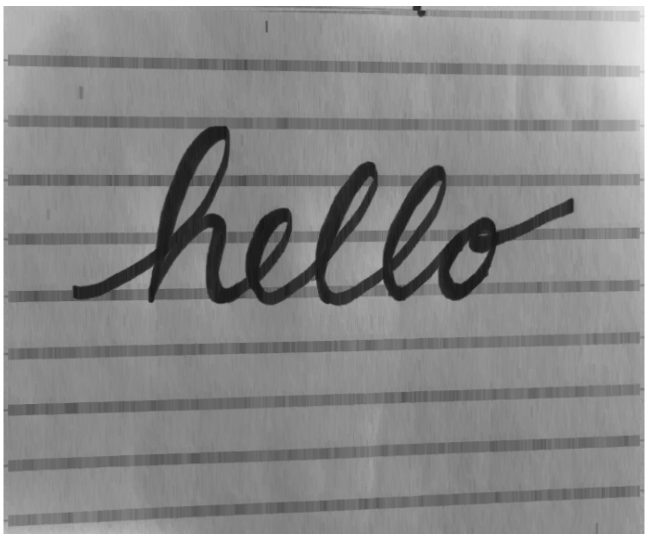

*Figure 42: 11 * 11 Max/Min Filter*

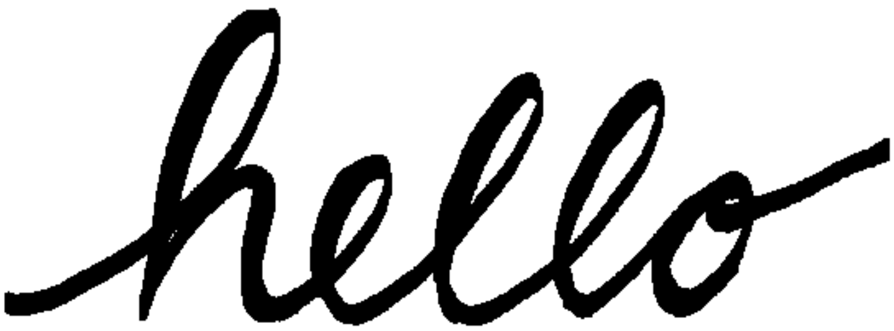

Figure 43: Max/Min Filter Thresholded 20%

The result of the Max filter on the sample image shows the thickening of the notepad lines and distortion of the pen marks.