<u>Lab 3 Report</u>

Author: *Pablo Ruiz*

Due Date: *11/23/2022 5:00 PM*

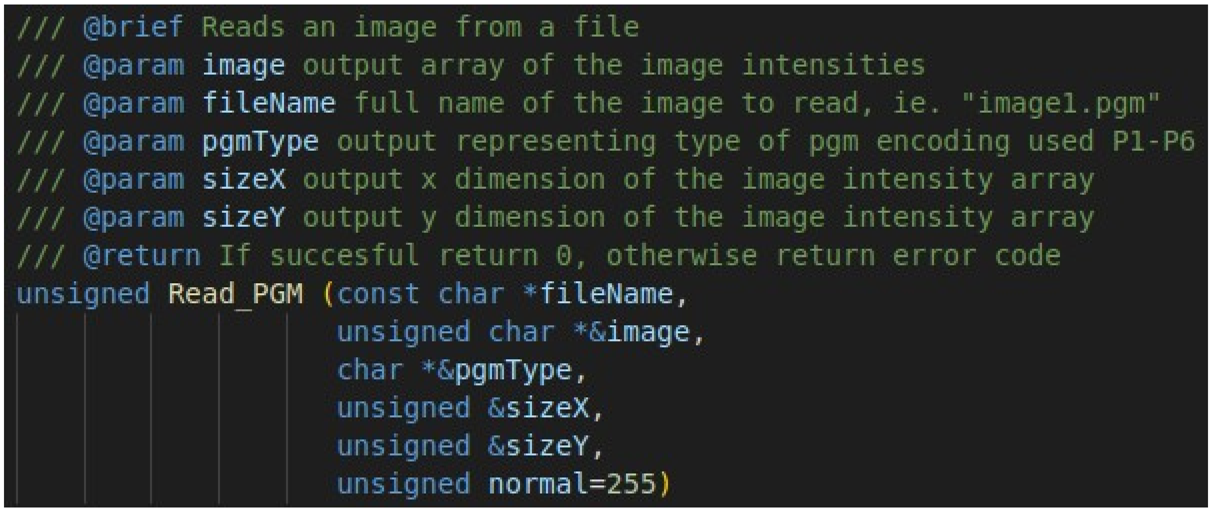
Objective:

The objective of this lab is to demonstrate the operation of the Sobel edge detection kernels on clean images and on images that have had noise artificially added to them. Additionally, the effects of applying a 3x3 Gaussian low pass (smoothing) kernel to the noisy image before applying Sobel edge detection will be compared. In order to compare the edge detection outputs from the three images (clean, noisy, and smooth) directly the output from the Sobel convolution will be converted to a binary image by applying a threshold to the raw values. Any magnitude values above the threshold will be mapped to 255 (white) while any values equal to or lower than the threshold will be mapped to 0. The criteria for comparison will be the percentage difference between the binary outputs for the noisy and smooth images and the benchmark which in this case is the binary output from the clean image.

Algorithms / Functions:

Read_PGM:

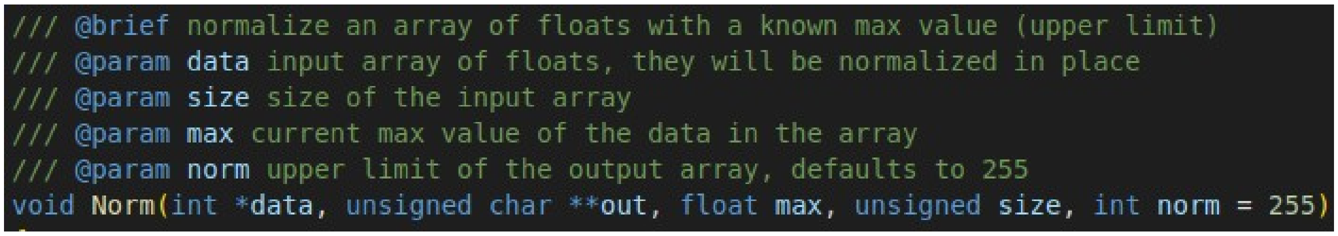
- Reads a pgm image from a file into a 1D array.



```
/// @brief Reads an image from a file
/// @param image output array of the image intensities
/// @param fileName full name of the image to read, ie. "image1.pgm"
/// @param pgmType output representing type of pgm encoding used P1-P6
/// @param sizeX output x dimension of the image intensity array
/// @param sizeY output y dimension of the image intensity array
/// @return If succesful return 0, otherwise return error code
unsigned Read_PGM (const char *fileName,
                   unsigned char *&image,
                   char *&pgmType,
                   unsigned &sizeX,
                   unsigned &sizeY,
                   unsigned normal=255)
```

Norm:

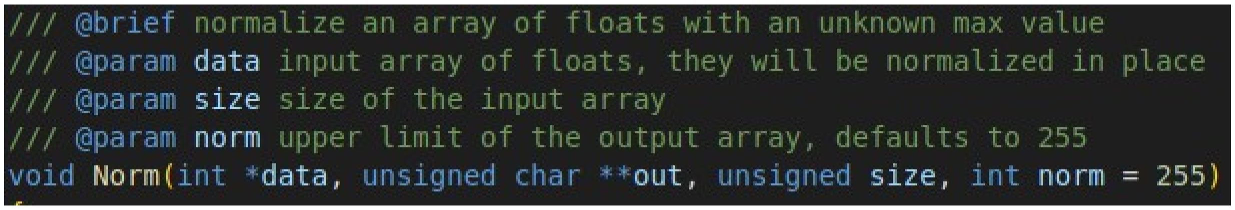
- Normalizes a 1D array taking a known max value as an argument.



```
/// @brief normalize an array of floats with a known max value (upper limit)
/// @param data input array of floats, they will be normalized in place
/// @param size size of the input array
/// @param max current max value of the data in the array
/// @param norm upper limit of the output array, defaults to 255
void Norm(int *data, unsigned char **out, float max, unsigned size, int norm = 255)
```

Norm (Overload):

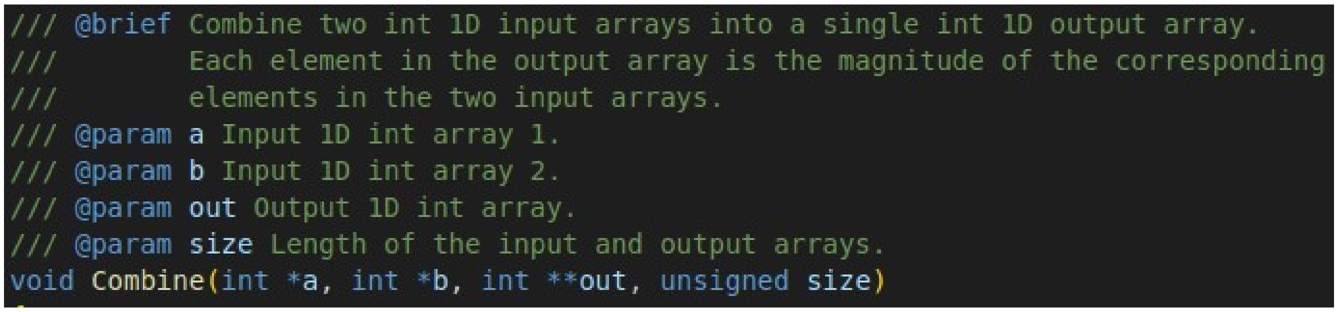
- Normalizes a 1D array with no known max value. Finds the max value within the array and uses that as its max value.



```
/// @brief normalize an array of floats with an unknown max value
/// @param data input array of floats, they will be normalized in place
/// @param size size of the input array
/// @param norm upper limit of the output array, defaults to 255
void Norm(int *data, unsigned char **out, unsigned size, int norm = 255)
```

Combine:

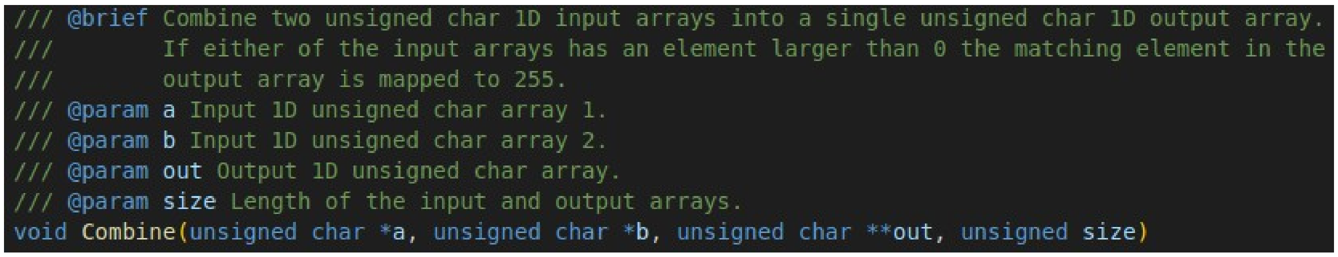
- Combines two 1D input arrays into a single 1D output array by obtaining the magnitude of the corresponding members of the input arrays.



```
/// @brief Combine two int 1D input arrays into a single int 1D output array.
///        Each element in the output array is the magnitude of the corresponding
///        elements in the two input arrays.
/// @param a Input 1D int array 1.
/// @param b Input 1D int array 2.
/// @param out Output 1D int array.
/// @param size Length of the input and output arrays.
void Combine(int *a, int *b, int **out, unsigned size)
```

Combine (Overload):

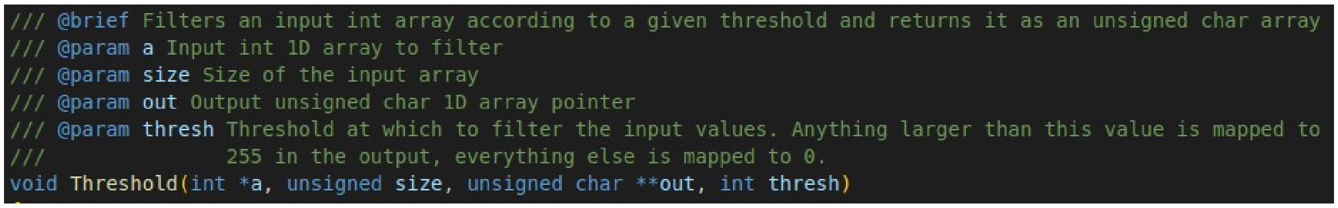
- Combines two 1D input arrays into a single 1D output array by using the corresponding elements of the input arrays as binary inputs in an OR statement. If either is non zero the corresponding value in the output array is 255, otherwise it is zero.



```
/// @brief Combine two unsigned char 1D input arrays into a single unsigned char 1D output array.
///        If either of the input arrays has an element larger than 0 the matching element in the
///        output array is mapped to 255.
/// @param a Input 1D unsigned char array 1.
/// @param b Input 1D unsigned char array 2.
/// @param out Output 1D unsigned char array.
/// @param size Length of the input and output arrays.
void Combine(unsigned char *a, unsigned char *b, unsigned char **out, unsigned size)
```

Threshold:

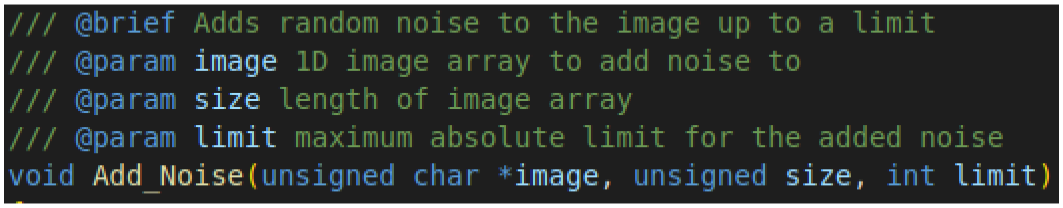
- Filters an input array into binary values (255 or 0) depending on if each corresponding input value is larger than a given threshold.



```
/// @brief Filters an input int array according to a given threshold and returns it as an unsigned char array
/// @param a Input int 1D array to filter
/// @param size Size of the input array
/// @param out Output unsigned char 1D array pointer
/// @param thresh Threshold at which to filter the input values. Anything larger than this value is mapped to
///               255 in the output, everything else is mapped to 0.
void Threshold(int *a, unsigned size, unsigned char **out, int thresh)
```

Add_Noise:

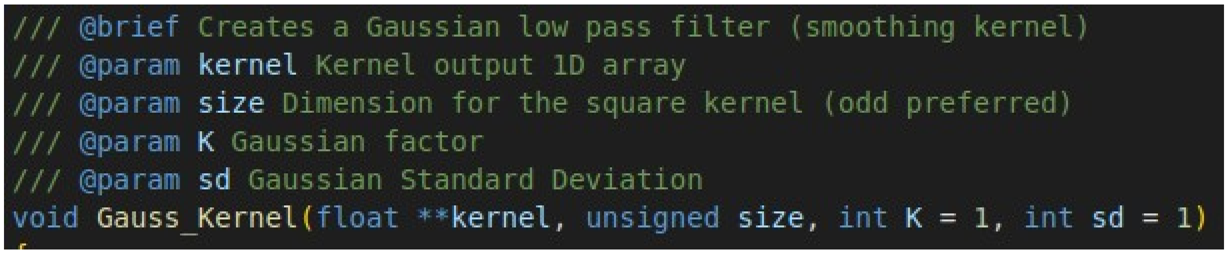
- Adds random noise to each element in a 1D array with a specified noise upper and lower limit passed as an argument.



```
/// @brief Adds random noise to the image up to a limit
/// @param image 1D image array to add noise to
/// @param size length of image array
/// @param limit maximum absolute limit for the added noise
void Add_Noise(unsigned char *image, unsigned size, int limit)
```

Gauss_Kernel:

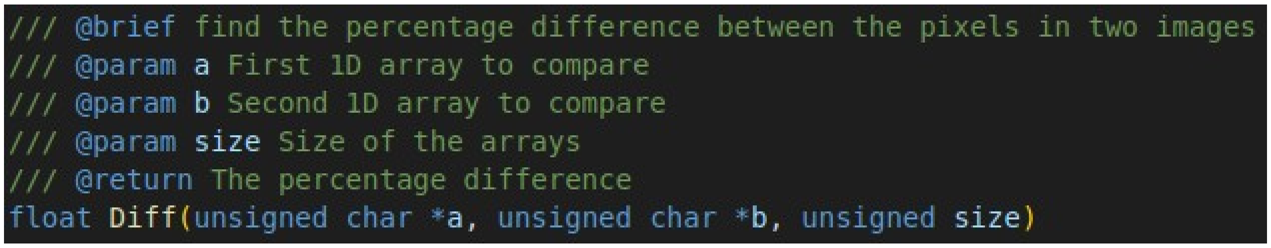
- Generates a Gaussian kernel of the specified size using the specified K and standard deviation as a 1D array.



```
/// @brief Creates a Gaussian low pass filter (smoothing kernel)
/// @param kernel Kernel output 1D array
/// @param size Dimension for the square kernel (odd preferred)
/// @param K Gaussian factor
/// @param sd Gaussian Standard Deviation
void Gauss_Kernel(float **kernel, unsigned size, int K = 1, int sd = 1)
```

Diff:

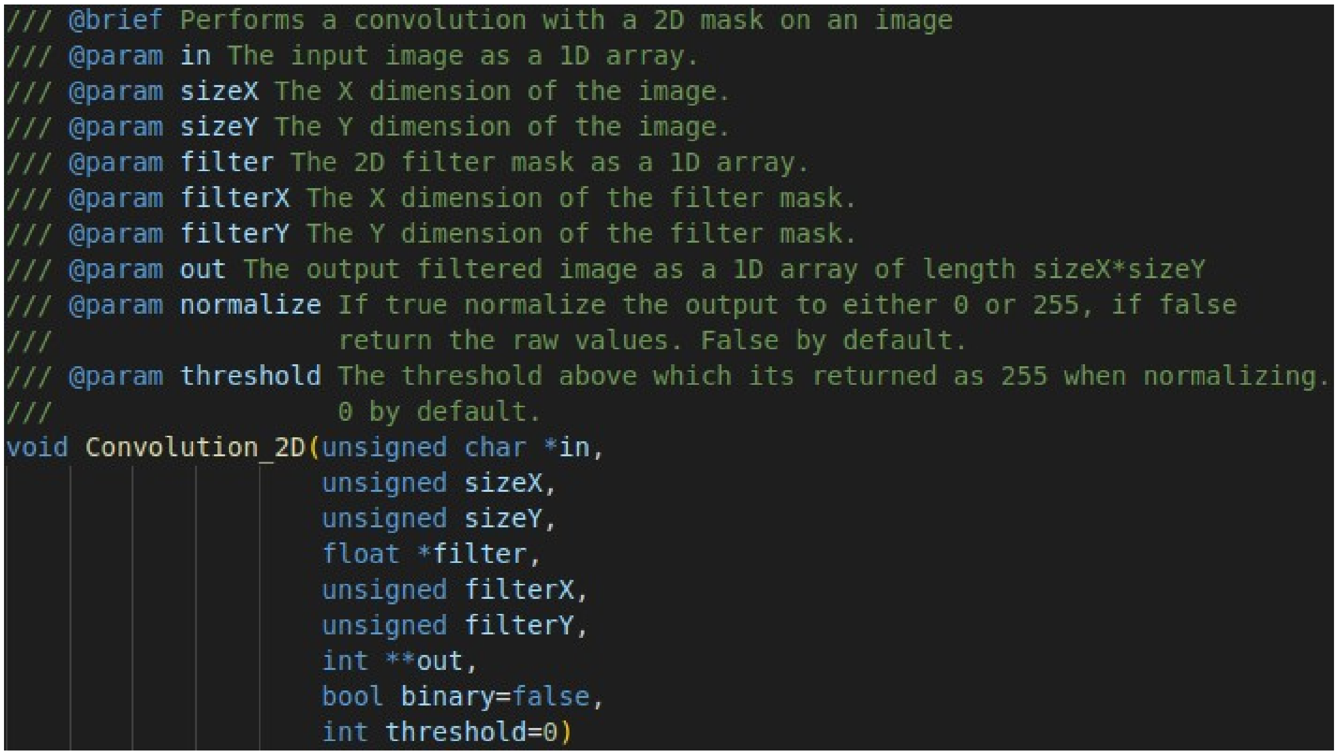
- Finds the how many elements are different between two arrays as a percentage of their total length.



```
/// @brief find the percentage difference between the pixels in two images
/// @param a First 1D array to compare
/// @param b Second 1D array to compare
/// @param size Size of the arrays
/// @return The percentage difference
float Diff(unsigned char *a, unsigned char *b, unsigned size)
```

Convolution_2D:

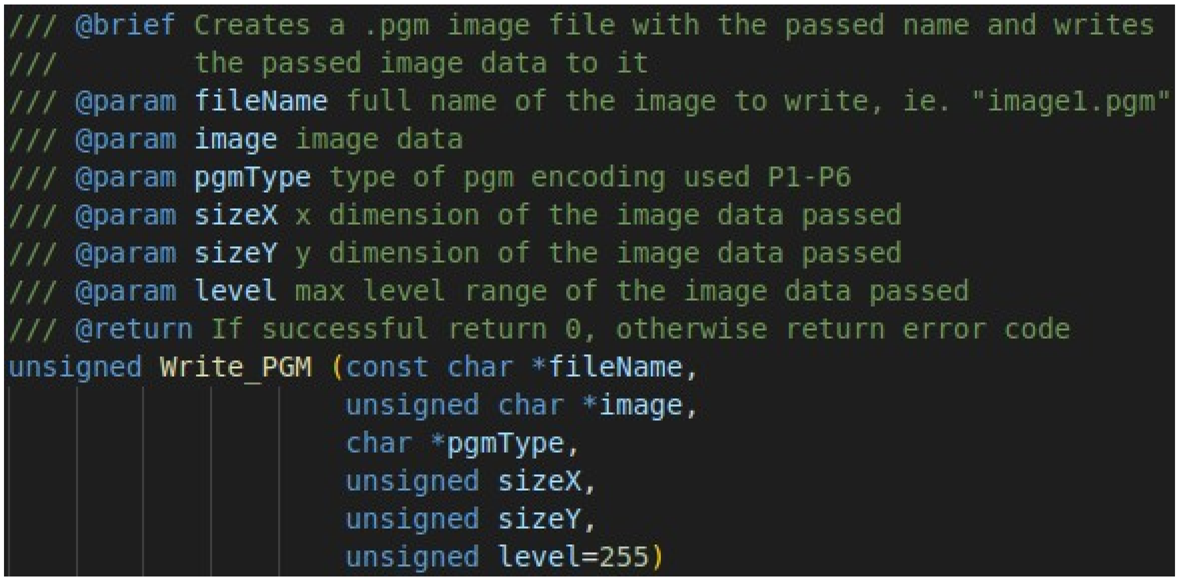
- Performs convolution on an input array using an input 2D kernel. Both the array and the kernel are assumed to be 2D arrays represented as a 1D array.



```
/// @brief Performs a convolution with a 2D mask on an image
/// @param in The input image as a 1D array.
/// @param sizeX The X dimension of the image.
/// @param sizeY The Y dimension of the image.
/// @param filter The 2D filter mask as a 1D array.
/// @param filterX The X dimension of the filter mask.
/// @param filterY The Y dimension of the filter mask.
/// @param out The output filtered image as a 1D array of length sizeX*sizeY
/// @param normalize If true normalize the output to either 0 or 255, if false
///                  return the raw values. False by default.
/// @param threshold The threshold above which its returned as 255 when normalizing.
///                  0 by default.
void Convolution_2D(unsigned char *in,
                    unsigned sizeX,
                    unsigned sizeY,
                    float *filter,
                    unsigned filterX,
                    unsigned filterY,
                    int **out,
                    bool binary=false,
                    int threshold=0)
```

Write_PGM:

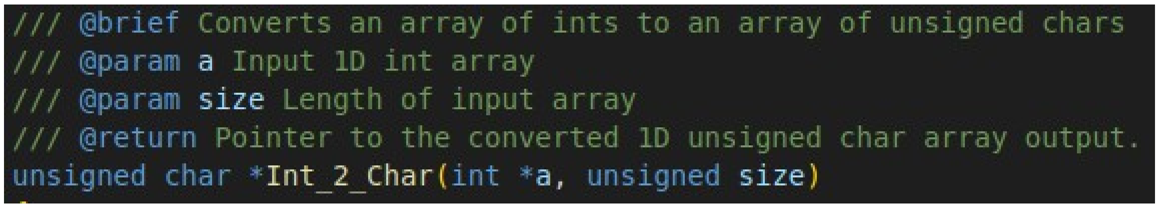
- Saves an input 1D array as a .pgm image in the local firectory with the name and dimensions passed as arguments.



```
/// @brief Creates a .pgm image file with the passed name and writes
///        the passed image data to it
/// @param fileName full name of the image to write, ie. "image1.pgm"
/// @param image image data
/// @param pgmType type of pgm encoding used P1-P6
/// @param sizeX x dimension of the image data passed
/// @param sizeY y dimension of the image data passed
/// @param level max level range of the image data passed
/// @return If successful return 0, otherwise return error code
unsigned Write_PGM (const char *fileName,
                   unsigned char *image,
                   char *pgmType,
                   unsigned sizeX,
                   unsigned sizeY,
                   unsigned level=255)
```

Int_2_Char:

- Converts a 1D input int array into a 1D unsigned char array.

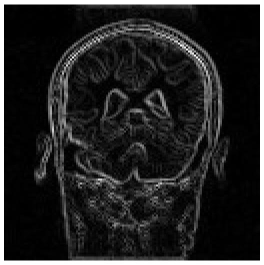


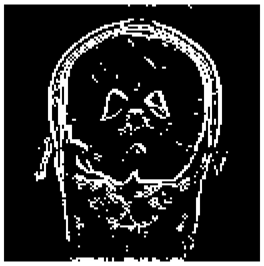
```
/// @brief Converts an array of ints to an array of unsigned chars
/// @param a Input 1D int array
/// @param size Length of input array
/// @return Pointer to the converted 1D unsigned char array output.
unsigned char *Int_2_Char(int *a, unsigned size)
```

Result:

Mri.pgm Results (clean image edge detection only):

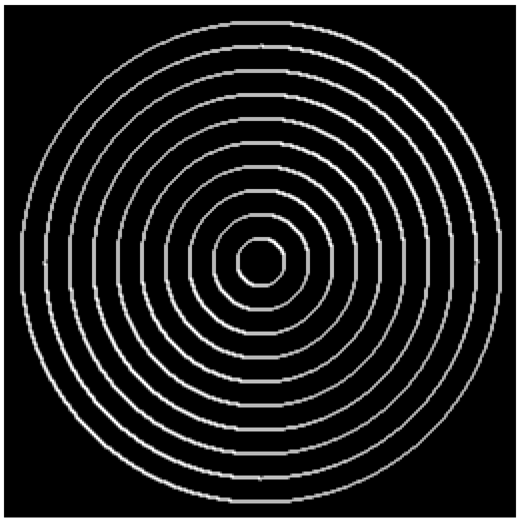
- Threshold_Clean = 150

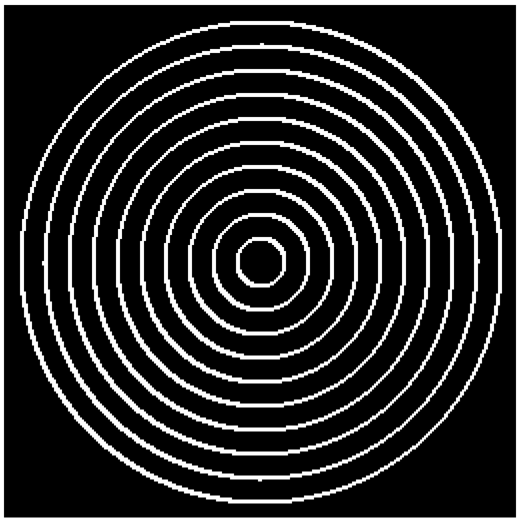

Sobel Raw Clean Image

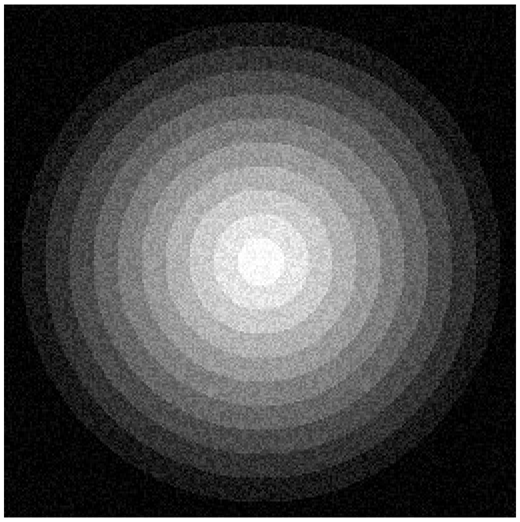

Binary Clean Image

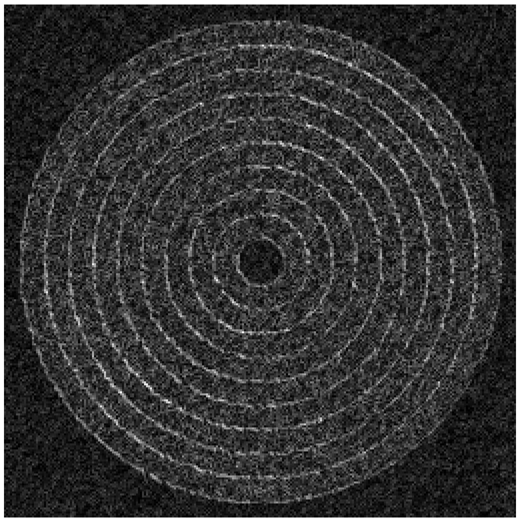
Cake.pgm Results:

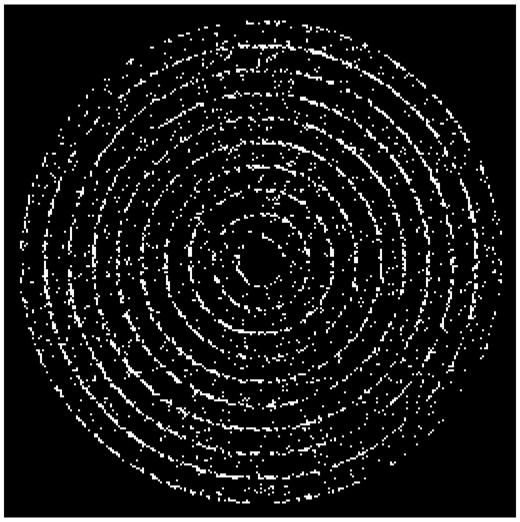
- Threshold_Clean = 0, Threshold_Noisy = 90, Threshold_Smooth = 35, Noise_Limit = 20
  - Difference between clean and noisy binary images: 10.3561%
  - Difference between clean and smooth binary images: 7.96356%

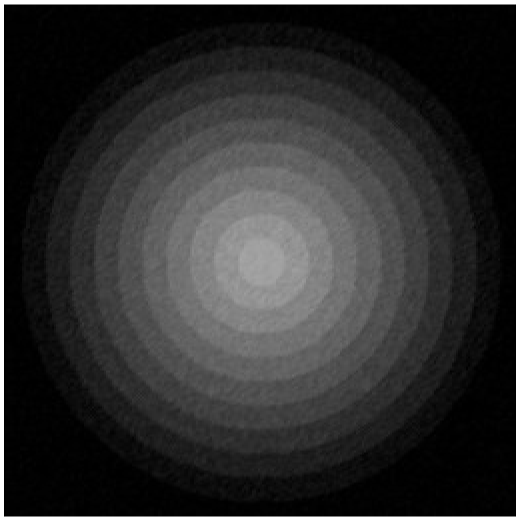

Sobel Raw Clean Image

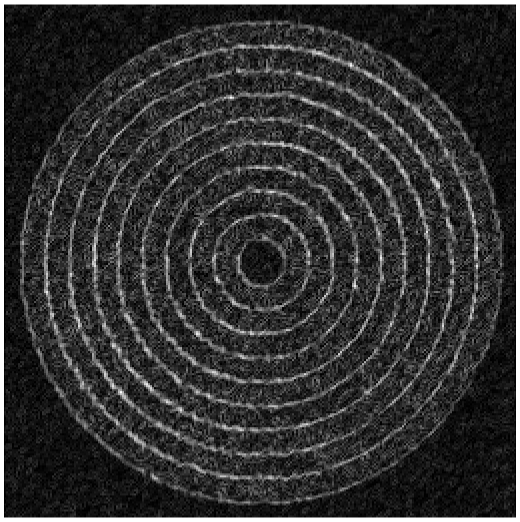

Binary Clean Image

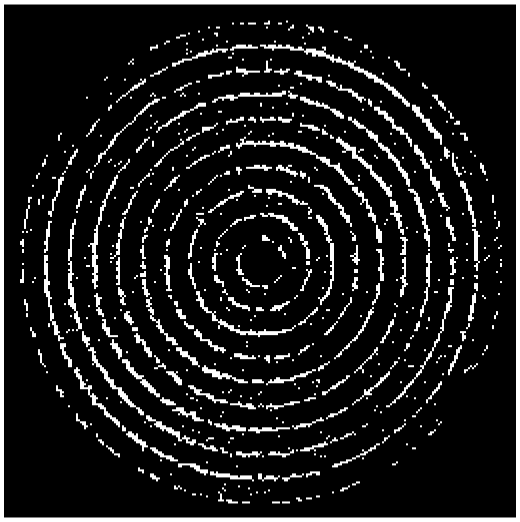

Source Image After Artificial Noise is Added


Sobel Raw Noisy Image


Binary Noisy Image


Source Image After Artificial Noise is Added and Smoothing Filter is Applied


Sobel Raw Smooth Image


Binary Smooth Image

Noise Level Comparison Chart:

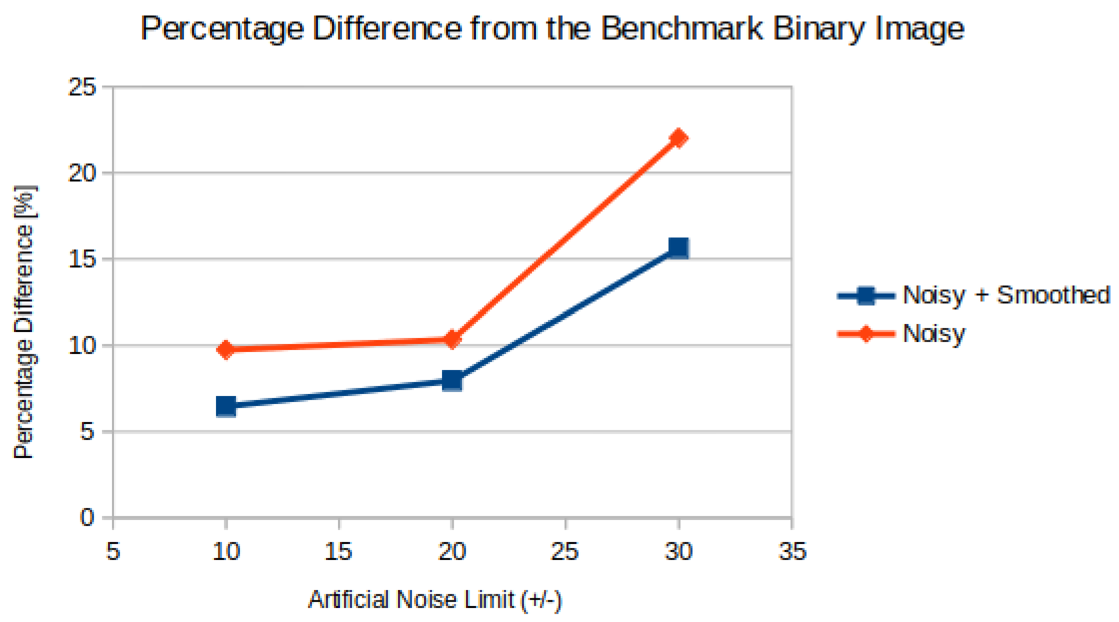


Chart 1.

- As can be observed in the chart above the percentage difference is always lower for the smoothed image (which means its closer to the benchmark). Additionally, it is also clear that the difference increases exponentially as the artificial noise limit increases.
- The above results were obtained using the following constant parameters:
  - Threshold_Clean = 0
  - Threshold_Noisy = 90
  - Threshold_Smooth = 35

Observations:

The first critical observation made while developing the code for this lab was that the sign of the output values from the Sobel convolution is not important, what matters is their magnitude, because for edge detection in the spatial domain with a kernel the absolute rate of change is more important than the direction in which that rate of change occurs. For this reason the absolute value of the Sobel convolution outputs was used when creating a binary image and applying a threshold.

To obtain the best results from the edge detection on each of the binary images it was necessary to apply different thresholds for each. This is because in the case of the noisy image it is important to raise the threshold in order to remove as much noise as possible while trying to minimize the effect on the edges. On the other hand, in the case of the smooth image, much of the noise was removed or at least reduced in magnitude by the smoothing filter. This also affected the magnitude of the rate of change on the edges, therefore the threshold can and should be lower in order to preserve more of the edges.

In the case of the cake image, small percentage differences between the outputs of the benchmark (clean image) and the noisy or smooth outputs are significant since the edges comprise a small part of the total pixel count in these images. Therefore the small 2-3% that the smooth binary output was closer to the benchmark means a large portion of the noise artifacts were eliminated.

After applying different size Gaussian smoothing filters to the noisy image and observing the effects on the smooth binary output it was clear that while smoothing can help revert the effects of the added noise, too much smoothing can be a bad thing. When using a 5x5 Gaussian filter, which is close to the theoretical smoothing limit for the selected standard deviation (σ=1) which would be 6σx6σ, the whole image had lost a lot of intensity and the edges were very vague. This did remove most if not all of the noise but it also made the edges very hard to detect and overall the results were worse than when using a smaller smoothing kernel. Ultimately, a 3x3 Gaussian smoothing kernel with K=1 and σ=1 was chosen as the best balance of attenuating noise while preserving the original edges of the image.

When comparing the final result it is clear that after the noise was added it was impossible to remove all of it from the binary noisy output. Due to the noise being added at each individual pixel randomly it created relatively large and sudden changes in intensity from pixel to pixel. These translated into large rates of change in the Sobel filter raw output. To filter out all of these rate of change spikes from the binary noisy output a large threshold would be necessary.

Such a large threshold would inevitably also filter out a significant part of the real edges that were detected. Thus a more accurate result (lower percentage difference) is obtained by leaving some of the noise in the final image and preserving as much of the edges as possible. This is to say, there is a threshold band that will return the smallest difference to the original, if its too low too much noise is let through while if its too high too much of the edges is filtered out. Applying a smoothing filter after the noise has been added does clearly improve the accuracy of the final smooth binary output with the caveats mentioned earlier. It does this by attenuating the noise more relative to the edges and creating a bigger delta between the threshold at which a significant part of the noise is removed and the threshold at which a significant part of the edges is also removed. It is even possible that a purpose tuned smoothing filter could attenuate the noise enough to generate a binary output almost identical to the original output. However, this depends of the image size, edge clarity, and noise level, on a case by case basis and there is no one size fits all solution. As seen in chart 1, the percentage difference was always lower for the smoothed image and it increased exponentially as the noise level limit was increased.

Conclusions:

In conclusion, the Sobel edge detection kernels in the spatial domain is a relatively effective and computationally efficient way to perform edge detection on images. However, image noise or interference can pose a severe challenge to the Sobel filter and can result in a significantly degraded and noisy final result. A simple at least partial remedy to noisy images is to apply some type of smoothing filter before applying the Sobel filter. The smoothing filter must be strong enough to reduce the intensity of the noise while not diminishing the gradient changes at the edges to the point where they can no longer be detected. This will improve the quality of the final result but will likely not completely undo the effects of the noise.

Source Code:

```
/*
Computer Vision
```

```
Assignment 3
Author: Pablo Ruiz
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <iostream>

/// @brief Reads an image from a file
/// @param image output array of the image intensities
/// @param fileName full name of the image to read, ie. "image1.pgm"
/// @param pgmType output representing type of pgm encoding used P1-P6
/// @param sizeX output x dimension of the image intensity array
/// @param sizeY output y dimension of the image intensity array
/// @return If succesful return 0, otherwise return error code
unsigned Read_PGM (const char *fileName,
unsigned char *&image,
char *&pgmType,
unsigned &sizeX,
unsigned &sizeY,
unsigned normal=255)
{
unsigned levels, size;
pgmType = (char*)malloc(sizeof(char) * 3);

// Read Image
FILE *iFile = fopen(fileName,"r");
if(iFile == 0) return 1; //return read failure error code
if(4 != fscanf(iFile, "%s %u %u %u ", pgmType, &sizeX, &sizeY, &levels)) return 2; //return
scan error code
size = sizeX * sizeY;
image = (unsigned char *) malloc(size);
fread(image,sizeof(unsigned char),size,iFile);
fclose(iFile);
return 0;
}


/// @brief normalize an array of floats with a known max value (upper limit)
/// @param data input array of floats, they will be normalized in place
/// @param size size of the input array
/// @param max current max value of the data in the array
/// @param norm upper limit of the output array, defaults to 255
void Norm(int *data, unsigned char **out, float max, unsigned size, int norm = 255)
{
// Normalize intensity values
*out = (unsigned char*)calloc(sizeof(unsigned char), size);
float factor = norm / max;
for (unsigned i = 0; i < size; ++i)
{
(*out)[i] = data[i] * factor;
}
}

/// @brief normalize an array of floats with an unknown max value
/// @param data input array of floats, they will be normalized in place
/// @param size size of the input array
/// @param norm upper limit of the output array, defaults to 255
void Norm(int *data, unsigned char **out, unsigned size, int norm = 255)
{
float max = data[0];
//find max
for (unsigned i = 0; i < size; ++i)
{
max = data[i] > max ? data[i] : max;
}
Norm(data, out, max, size, norm);
}

/// @brief Combine two int 1D input arrays into a single int 1D output array.
/// Each element in the output array is the sum of the corresponding elements in the two input
arrays.
/// @param a Input 1D int array 1.
/// @param b Input 1D int array 2.
/// @param out Output 1D int array.
/// @param size Length of the input and output arrays.
void Combine(int *a, int *b, int **out, unsigned size)
{
*out = (int*)calloc(sizeof(int), size);
for (unsigned i = 0; i < size; ++i)
{
(*out)[i] = sqrt(pow(a[i],2) + pow(b[i],2));
}
}
```

```
/// @brief Combine two unsigned char 1D input arrays into a single unsigned char 1D output
array.
/// If either of the input arrays has an element larger than 0 the matching element in the
output array is mapped to 255.
/// @param a Input 1D unsigned char array 1.
/// @param b Input 1D unsigned char array 2.
/// @param out Output 1D unsigned char array.
/// @param size Length of the input and output arrays.
void Combine(unsigned char *a, unsigned char *b, unsigned char **out, unsigned size)
{
*out = (unsigned char*)calloc(sizeof(unsigned char), size);
for (unsigned i = 0; i < size; ++i)
{
(*out)[i] = (b[i] || b[i]) ? 255 : 0;
}

}

/// @brief Filters an input int array according to a given threshold and returns it as an
unsigned char array
/// @param a Input int 1D array to filter
/// @param size Size of the input array
/// @param out Output unsigned char 1D array pointer
/// @param thresh Threshold at which to filter the input values. Anything larger than this value
is mapped to
/// 255 in the output, everything else is mapped to 0.
void Threshold(int *a, unsigned size, unsigned char **out, int thresh)
{
*out = (unsigned char*)calloc(sizeof(unsigned char), size);
for (unsigned i = 0; i < size; ++i)
{
(*out)[i] = a[i] > thresh ? 255 : 0;
}
}

/// @brief Converts an array of ints to an array of unsigned chars
/// @param a Input 1D int array
/// @param size Length of input array
/// @return Pointer to the converted 1D unsigned char array output.
unsigned char *Int_2_Char(int *a, unsigned size)
{
unsigned char* out = (unsigned char*)calloc(sizeof(unsigned char), size);
for (unsigned i = 0; i < size; ++i)
{
```

```
out[i] = a[i] > 255 ? 255 : (a[i] < 0 ? 0 : a[i]);
}
return out;
}

/// @brief Creates a .pgm image file with the passed name and writes
/// the passed image data to it
/// @param fileName full name of the image to write, ie. "image1.pgm"
/// @param image image data
/// @param pgmType type of pgm encoding used P1-P6
/// @param sizeX x dimension of the image data passed
/// @param sizeY y dimension of the image data passed
/// @param level max level range of the image data passed
/// @return If successful return 0, otherwise return error code
unsigned Write_PGM (const char *fileName,
unsigned char *image,
char *pgmType,
unsigned sizeX,
unsigned sizeY,
unsigned level=255)
{
//Write image to file
FILE *iFile = fopen(fileName,"w");
if(iFile==0) return 1; //return error code failed to open
fprintf(iFile, "%s\n%u %u\n%u\n", pgmType,sizeX,sizeY,level);//write header
fwrite(image, sizeof(unsigned char), sizeX * sizeY, iFile); //write binary image
fclose(iFile);
return 0;
}

/// @brief Adds random noise to the image up to a limit
/// @param image 1D image array to add noise to
/// @param size length of image array
/// @param limit maximum absolute limit for the added noise
void Add_Noise(unsigned char *image, unsigned size, int limit)
{
float noisyPixel, noise;
for (unsigned i = 0; i < size; ++i)
{
noise = ((2*(float)rand() - RAND_MAX) / RAND_MAX) * limit;
noisyPixel = noise + image[i];
if (noisyPixel > 255)
image[i] = 255;
else if (noisyPixel < 0)
```

```
image[i] = 0;
else
image[i] = noisyPixel;
}
}

/// @brief find the percentage difference between the pixels in two images
/// @param a First 1D array to compare
/// @param b Second 1D array to compare
/// @param size Size of the arrays
/// @return The percentage difference
float Diff(unsigned char *a, unsigned char *b, unsigned size)
{
unsigned diffCount = 0;
for (unsigned i = 0; i < size; ++i)
{
a[i] != b[i] ? ++diffCount : 0;
}
return ((float)diffCount/size) * 100;
}

/// @brief Creates a Gaussian low pass filter (smoothing kernel)
/// @param kernel Kernel output 1D array
/// @param size Dimension for the square kernel (odd preferred)
/// @param K Gaussian factor
/// @param sd Gaussian Standard Deviation
void Gauss_Kernel(float **kernel, unsigned size, int K = 1, int sd = 1)
{
unsigned index;
float sum = 0;
*kernel = (float *)calloc(sizeof(float), size*size);

//Calculate the factor for every element of the filter
for (unsigned i = 0; i < size; ++i)
{
for (unsigned j = 0; j < size; ++j)
{
index = i * size + j;
(*kernel)[index] = (float)K * exp(-(pow(i,2)+pow(j,2))/((float)2*pow(sd,2)));
sum += (*kernel)[index];
}

}
```

```
//Divide every element by the total so that together they all add up to one
for (unsigned i = 0; i < size*size; ++i)
{
(*kernel)[i] /= sum;
}

}

/// @brief Performs a convolution with a 2D mask on an image
/// @param in The input image as a 1D array.
/// @param sizeX The X dimension of the image.
/// @param sizeY The Y dimension of the image.
/// @param filter The 2D filter mask as a 1D array.
/// @param filterX The X dimension of the filter mask.
/// @param filterY The Y dimension of the filter mask.
/// @param out The output filtered image as a 1D array of length sizeX*sizeY
/// @param normalize If true normalize the output to either 0 or 255, if false return the raw
values. False by default.
/// @param threshold The threshold above which its returned as 255 when normalizing. 0 by
default.
/// @return
void Convolution_2D(unsigned char *in, unsigned sizeX, unsigned sizeY, float *filter, unsigned
filterX, unsigned filterY, int **out, bool binary=false, int threshold=0)
{
unsigned index,
indexFilter,
centerX = filterX/2,
centerY = filterY/2;
int filteredOrigin,
indexCorrected,
offsetX,
offsetY,
max;

*out = (int*)calloc(sizeof(int), sizeX*sizeY); //allocate space for the output array

for (unsigned i = 0; i < sizeY; ++i)
{
for (unsigned j = 0; j < sizeX; ++j)
{
index = sizeX * i + j; //calculate the index for the current image intensity value
filteredOrigin = 0; //reset the output value for the current origin
for (unsigned k = 0; k < filterY; ++k)
{
```

```
for (unsigned l = 0; l < filterX; ++l)
{
indexFilter = filterX * k + l; //calculate the index for the current filter value
offsetX = l - centerX; //calculate the X offset relative to the center of the filter
offsetY = k - centerY; //calculate the Y offset relative to the center of the filter
indexCorrected = index + offsetX + offsetY*sizeX;

// If the corrected index is:
// -negative
// -larger than the end of the array
// -larger than the current end of row
// -smaller than the beginning of current row
// Add 0 otherwise add the product of filter times the intensity
if (
indexCorrected < 0 ||
indexCorrected > (int)(sizeX * sizeY) ||
(l > centerX ? indexCorrected > (int)((i+1)*sizeX) : false) ||
(l < centerX ? indexCorrected < (int)(i*sizeX) : false)
)
filteredOrigin += 0;
else
filteredOrigin += filter[indexFilter] * in[indexCorrected];

}
}
max = (filteredOrigin > max) ? filteredOrigin : max;
(*out)[index] = (binary) ? ((abs(filteredOrigin) > threshold) ? 255 : 0) : abs(filteredOrigin); //if
binary is enabled and the output is above the threshhold save as 255
}
}
}

int main(int argc, char *argv[]){

unsigned sizeX, sizeY, size; //image width, height, size
unsigned char *image; //image array
char *pgmType;
char *nameRead = argv[1];
int threshold = atoi(argv[2]);
int thresholdNoisy = atoi(argv[3]);
int thresholdSmooth = atoi(argv[4]);
int limit = atoi(argv[5]);
int *filteredX, *filteredY, *filteredFull, *noisyFull, *smoothFull, *imageSmooth;
unsigned char *noisyChar, *filteredChar, *smoothChar;
```

```
float sobelFilterX[] = {1, 0, -1, 2, 0, -2, 1, 0, -1};
float sobelFilterY[] = {1, 2, 1, 0, 0, 0, -1, -2, -1};
float *smoothFilter;// = {0.0751, 0.1238, 0.0751, 0.1238, 0.2042, 0.1238, 0.0751, 0.1238, 
0.0751};

//Read image and normalize the values
Read_PGM(nameRead, image, pgmType, sizeX, sizeY);
size = sizeX*sizeY;

#pragma region Clean
//Apply sobel filter and return filtered normalized array
Convolution_2D(image, sizeX, sizeY, sobelFilterX, 3, 3, &filteredX, false);
Convolution_2D(image, sizeX, sizeY, sobelFilterY, 3, 3, &filteredY, false);

//Combine X and Y filtered edge images
Combine(filteredX, filteredY, &filteredFull, size);

//Normalize filtered results
Norm(filteredFull, &filteredChar, size);

//Save the filtered edge image
Write_PGM("Sobel_Clean.pgm", filteredChar, pgmType, sizeX, sizeY);

//Threshold filtered results
Threshold(filteredFull, size, &filteredChar, threshold);

//Save the binary edge image
Write_PGM("Binary_Clean.pgm", filteredChar, pgmType, sizeX, sizeY);
#pragma endregion

#pragma region Noise
//Add noise to image
Add_Noise(image, size, limit);

//Save noisy image for reference
Write_PGM("Source_Noisy.pgm", image, pgmType, sizeX, sizeY);

//Apply sobel filter to noisy image
Convolution_2D(image, sizeX, sizeY, sobelFilterX, 3, 3, &filteredX);
Convolution_2D(image, sizeX, sizeY, sobelFilterY, 3, 3, &filteredY);

//Combine X and Y filtered edge images
Combine(filteredX, filteredY, &noisyFull, size);
```

```
//Normalize filtered results
Norm(noisyFull, &noisyChar, size);

//Save the filtered edge image
Write_PGM("Sobel_Noisy.pgm", noisyChar, pgmType, sizeX, sizeY);

//Threshold filtered results
Threshold(noisyFull, size, &noisyChar, thresholdNoisy);

//Save the binary edge image
Write_PGM("Binary_Noisy.pgm", noisyChar, pgmType, sizeX, sizeY);
#pragma endregion

#pragma region Smooth
//Smooth
Gauss_Kernel(&smoothFilter, 3);
Convolution_2D(image, sizeX, sizeY, smoothFilter, 3, 3, &imageSmooth);

//Replace original image with smoothed image
image = Int_2_Char(imageSmooth, size);

//Save smooth image for reference
Write_PGM("Source_Smooth.pgm", image, pgmType, sizeX, sizeY);

//Apply sobel filter to noisy smoothed image
Convolution_2D(image, sizeX, sizeY, sobelFilterX, 3, 3, &filteredX);
Convolution_2D(image, sizeX, sizeY, sobelFilterY, 3, 3, &filteredY);

//Combine X and Y filtered edge images
Combine(filteredX, filteredY, &smoothFull, size);

//Normalize filtered results
Norm(smoothFull, &smoothChar, size);

//Save the filtered edge image
Write_PGM("Sobel_Smooth.pgm", smoothChar, pgmType, sizeX, sizeY);

//Threshold filtered results
Threshold(smoothFull, size, &smoothChar, thresholdSmooth);

//Save the binary edge image
Write_PGM("Binary_Smooth.pgm", smoothChar, pgmType, sizeX, sizeY);
#pragma endregion
```

```
//Print the percentage change between noisy and not
std::cout << "Difference between clean and noisy binary edges: " << Diff(filteredChar, noisyChar, size) << "%" << std::endl;
std::cout << "Difference between clean and smooth binary edges: " << Diff(filteredChar, smoothChar, size) << "%" << std::endl;

return 0;
}
```