# Introduction

This project is about performing 2D convolution of three different filters through an image. The three types of convolutions performed were standard 2D convolution through a filter, Sobel filter for edge detection, and correlation.

# 2D Convolution

The filter used for this convolution is a 5x5 filter h.

$$h = 1/81\begin{bmatrix}1 & 2 & 3 & 2 & 1\\ 2 & 4 & 6 & 4 & 2\\ 3 & 6 & 9 & 6 & 3\\ 2 & 4 & 6 & 4 & 2\\ 1 & 2 & 3 & 2 & 1\end{bmatrix}$$

Equation 1

The original image is a black and white image of a man in front of a bookcase [Figure 1]. This image is convolved with the filter h [Equation 1]. The resulting image [Figure 2] is a slightly blurry version of the original image. This filter takes the original pixel, and adds its value to other surrounding with certain weights. The original pixel has the greatest weight with the weight decreasing as you go out into the surrounding 5x5 pixels.

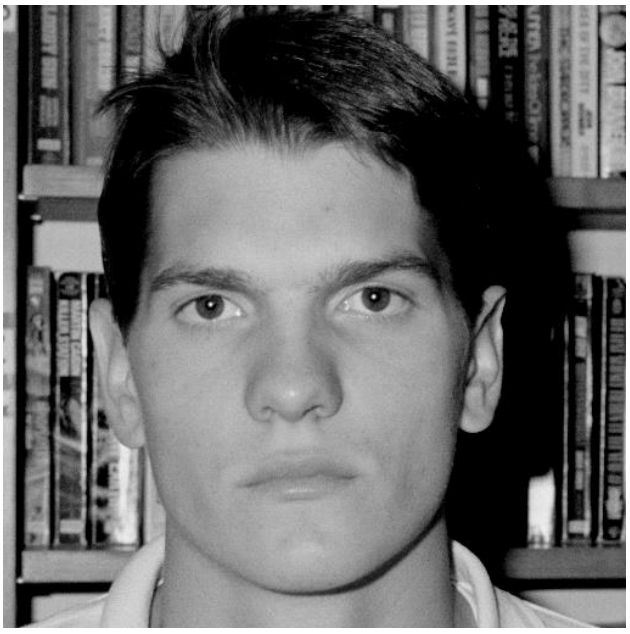

Figure 1 Original Image

Figure 2 Fuzzy Image

## Sobel Filter

The sobel filter is two filters S1 and S2 [Equation 2] convolved with the input image [Figure 1] and added together to create the output image [Figure 3]. These filters cause edge detection in the output image where white is where there is a contrast between neighboring pixels.

$$S1 = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad S2 = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Equation 2

Figure 3 Edge Detection

## Correlation

Correlation takes a small image [Figure 4] and compares it with the input image [Figure 1]. The output image [Figure 5] shows white where the two images are similar and black where they are different. First, the pixel values in the original image are squared and convolved with a filter of the same size as the input filter with all h[m,n]=1. The original input image is convolved with the original filter, and the individual pixels in the resulting image are divided by the pixel values in the previous image. The resulting pixels are then scaled to be between 0-255.

Figure 4 Filter Image

Figure 5 Correlation Image

## Conclusion

As we did a similar project in Discrete Signals, I was able use some of my old code. I updated most of the code to use a class structure and made general improvements to the code to make it more general for any .pgm file and filter. I did encounter some issues where after allocating memory and reading in the input file. I tried to return the pointer holding that data from a GetData() function, but after returning, I think garbage collection deallocated my array holding the data. I just learned I had to allocate in the caller function and then everything worked. I do think it is amazing how a simple matrix can modify an image like in the sobel filter to create edge filtering.

# Code Appendix

## main.cpp

```
#include "Convolution.h"
#include "pgmIO.h"


/**
 * \brief Multiplies the given matrix with a scalar
 * \param scalar
 * \param h matrix to multiply
 * \param m number of rows in matrix
 * \param n number of column in matrix
 */
void MultMatrixScalar(const double scalar, double *h, const int m, const int
n)
{
    for(auto i=0; i<m; i++)
    {
        for(auto j=0; j<n; j++)
        {
            h[i*m+j] = scalar * h[i*m+j];
        }
    }
}


int main(const int argc, char *argv[])
{
    if (argc < 3)
    {
        printf("Wrong number of arguments");
        printf("Format: .exe inFile outFile");
        exit(EXIT_FAILURE);
    }


    double h[] = {
        1,2,3,2,1,
        2,4,6,4,2,
        3,6,9,6,3,
        2,4,6,4,2,
        1,2,3,2,1
    };
    MultMatrixScalar(1 / 81.0f, h, 5, 5);


    //Convolution::Conv2D(argv[1], argv[2], h, 5, 5);
    //Convolution::Sobel(argv[1], argv[2]);
    Convolution::Correlate(argv[1], argv[2], argv[3]);


    return 0;
}
```

PgmIO.h

```
#pragma once
#include <cstdio>
#include <string>
#include <fstream>
#include <sstream>

#pragma warning(disable:4996)

using namespace std;

// ReSharper disable CppInconsistentNaming
class PgmIO
    // ReSharper restore CppInconsistentNaming
{
public:
    /**
     * \brief Attempts to open the given .pgm files and parses the header.
     * \param inFile Input .pgm
     * \param outFile Output .pgm
     */
    PgmIO(const string& inFile, const string& outFile = "noFile")
    {
        in.open(inFile, ios::in | ios::binary);
        if(outFile != "noFile")out.open(outFile, ios::binary | ios::out);

        if (!in.is_open())
        {
            printf("Error opening input file\n");
            exit(EXIT_FAILURE);
        }

        if (outFile != "noFile" && !out.is_open())
        {
            printf("error opening output file\n");
            exit(EXIT_FAILURE);
        }
        HeaderParser();


    }

    ~PgmIO()
    {
        in.close();
        out.close();
    }

    typedef struct
    {
        //2 char number to identify file type, pgm is P5
        string magicNumber;

        //comments starting each line with #
        string comments;
```

```
        // width of the image
        int width{};

        //height of the image
        int height{};

        //max gray value
        int maxVal{};

    } pgm_file_header;


    /**
     * \brief Parse the data of the input file into a vector. Must allocate
space for x before hand
     * \param x vector to store data into.
     * \param xOffSet offset for where to read data into x
     * \param yOffSet offset for where to read data into y
     */
    void GetData(double * x, const int xOffSet = 0, const int yOffSet = 0)
    {
        string ss;
        const auto temp = new unsigned char[dataSize];
        in.seekg(dataIndex, ios::beg);
        in.read(reinterpret_cast<char*>(temp), dataSize);
        //put image into padded x
        for (auto i = 0; i < header.height; i++)
        {
            for (auto j = 0; j < header.width; j++)
            {
                x[(i+xOffSet)*header.width + yOffSet+j] =
temp[i*header.width+j];
            }
        }
        delete[] temp;
    }

    /**
     * \brief Gets the header of the input file
     * \return header
     */
    pgm_file_header GetHeader() const
    {
        return header;
    }

    /**
     * \brief Writes data to output file .pgm
     * \param y Data vector to output
     * \param yHeader Header to use for .pgm
     * \param mOffset y row offset due to H
     * \param nOffset y col offset due to H
     */
    void WriteData(double *y, const pgm_file_header yHeader, const int
mOffset = 0, const int nOffset = 0)
    {
        string dim;
```

```
dim.assign(header.magicNumber).append("\n").append(header.comments).append("\
n");
        out.write(dim.c_str(), dim.length());
        dim.assign(std::to_string(header.width)).append("
").append(std::to_string(header.height)).append("\n");
        out.write(dim.c_str(), dim.length());
        dim.assign(std::to_string(header.maxVal)).append("\n");
        out.write(dim.c_str(), dim.length());

        for (auto i = 0; i < yHeader.width*yHeader.height; i++)
        {
            if (y[i] < 0) y[i] = 0;
            if (y[i] > header.maxVal) y[i] = header.maxVal;
        }

        auto* temp = new unsigned char[dataSize];

        for (auto i = 0; i < header.height; i++)
        {
            for(auto j=0; j < header.width; j++)
            {
                temp[i*header.width+j] =
static_cast<int>(floor(y[(i+mOffset)*yHeader.width+j+nOffset]));

            }
        }

        out.write(reinterpret_cast<char*>(temp), dataSize);


        out.close();
        delete[] temp;
    }


    int dataSize{};
private:
    ifstream in;
    ofstream out;
    pgm_file_header header;
    int dataIndex{};


    /**
     * \brief Parse the .pgm header and stores the data into the header var
     */
    void HeaderParser()
    {
        auto buffer = ReadLine();
        if (buffer != "P5")
        {
            printf("Invalid file format");
            exit(EXIT_FAILURE);
        }
        header.magicNumber = buffer;
```

```
        do
        {
            buffer = ReadLine();

            if (!buffer.find('#'))
header.comments.append(buffer).append("\n");
        } while (!buffer.find('#'));
        auto size = 0;
        header.width = stoi(Split(buffer, size)[0]);
        if(size>1)  header.height = stoi(Split(buffer, size)[1]);
        else
        {
            header.height = stoi(ReadLine());
        }
        header.maxVal = stoi(ReadLine());
        dataIndex = in.tellg();
        dataSize = header.width*header.height;
    }


    /**
     * \brief Split the given string into an array of strings
     * \param str string to parse
     * \param size size of the output array
     * \param delim delimiter to separate strings
     * \return array of strings. max size 10.
     */
    string* Split(const std::string& str, int &size, const char delim = ' ')
const
    {
        auto* cString = new string[10];
        stringstream ss(str);
        string token;
        auto i = 0;
        size = 0;
        while (getline(ss, token, delim))
        {
            cString[i++] = token;
            size++;
        }
        return cString;
    }

    /**
     * \brief Read in n bytes (unsigned char)
     * \param numBytes number of bytes to read in, default 1
     * \return c string with the given byte. Terminated in NULL
     */
    char* ReadByte(const int numBytes = 1)
    {
        const auto buffer = new char[numBytes + 1];
        in.read(buffer, numBytes);

        //string retVal(buffer, numBytes);
        buffer[numBytes] = NULL;
        return buffer;
    }
```

```
    /**
     * \brief Read in the line up until the delimiter
     * \param delim delimiter for when to stop looking. Default new line
     * \return The parsed line NOT including the delimiter
     */
    string ReadLine(const char delim = '\n')
    {
        string retVal, buffer;
        do
        {
            retVal.append(buffer);
            buffer.assign(ReadByte());
        } while (buffer.find(delim));
        return retVal;
    }

    /**
     * \brief Normalizes the upper and lower limits of the signal.
     * Anything lower than 0 becomes 0.
     * Anything higher than the header maxVal becomes maxVal
     * \param p Data to normalize
     */
    void NormalizeBounds(double* p) const
    {
        for (auto i = 0; i < dataSize; i++)
        {
            if (p[i] < 0) p[i] = 0;
            if (p[i] > header.maxVal) p[i] = header.maxVal;
        }
    }
};
```

Convolution.h

```
#pragma once
#include <string>
#include "pgmIO.h"
#include "Convolution.h"
#include <algorithm>
#include <iostream>

using namespace std;


class Convolution
{
public:
    Convolution() = default;

    /**
     * \brief 2D Convolve the given signal with a given h matrix. Writes out
the result to a file
     * \param inFile Input .pgm file
     * \param outFile Output .pgm file
     * \param h h matrix set up as a vector (h[x*numCol+y])
     * \param hM number of rows in h
     * \param hN number of columns in h
     */
    static void Conv2D(char* inFile, char* outFile, double* h, const int hM,
const int hN)
    {
        PgmIO pgmFile(inFile, outFile);


        // ReSharper disable CppInconsistentNaming
        const auto rowX = pgmFile.GetHeader().height,
                   colX = pgmFile.GetHeader().width,
                   rowZ = rowX + 2 * (hM - 1),
                   colZ = colX + 2 * (hN - 1),
                   rowY = rowX + hM - 1,
                   colY = colX + hN - 1;
        // ReSharper restore CppInconsistentNaming


        //Allocate and get data
        const auto y = static_cast<double*>(calloc(sizeof(double), rowY *
colY));
        const auto x = static_cast<double*>(calloc(sizeof(double), rowZ *
colZ));

        pgmFile.GetData(x, hM - 1, hN - 1);


        Convolve2D(y, rowY, colY, x, rowX, colX, h, hM, hN);
        auto yHeader = pgmFile.GetHeader();
        yHeader.width = colY;
        yHeader.height = rowY;

        //Output to file and cleanup
```

```
        pgmFile.WriteData(y, yHeader, hM - 2, hN - 2);

        free(y);
        free(x);
    }

    /**
     * \brief 2D Convolve the given signal with two matrices S1 and S2. The
output is
     * edge detection of the given image. Writes out the result to a file
     * \param inFile Input .pgm file
     * \param outFile Output .pgm file
     */
    static void Sobel(char* inFile, char* outFile)
    {
#define X(u,v) x[(u)*colX+(v)]
#define S1(u,v) s1[(u)*colH+(v)]
#define S2(u,v) s2[(u)*colH+(v)]
#define Y(u,v) y[(u)*colY+(v)]


        PgmIO pgmFile(inFile, outFile);


        // ReSharper disable CppInconsistentNaming
        const auto rowX = pgmFile.GetHeader().height,
                   colX = pgmFile.GetHeader().width,
                   rowH = 3,
                   colH = 3,
                   rowY = rowX +(rowH - 1),
                   colY = colX +colH - 1,
                   rowZ = rowX + 2 * (rowH - 1),
                   colZ = colX + 2 * (colH - 1);
        // ReSharper restore CppInconsistentNaming

        //Allocate and get data for x
        const auto y = static_cast<double*>(calloc(sizeof(double), rowY *
colY));
        const auto x = static_cast<double*>(calloc(sizeof(double), rowZ *
colZ));

        //zero pad the bounds of x
        pgmFile.GetData(x, rowH - 1, colH - 1);

        double s1[] = {
            1, 0, -1,
            2, 0, -2,
            1, 0, -1
        };
        double s2[] = {
            -1, -2, -1,
            0, 0, 0,
            1, 2, 1
        };

        //convolve with S1 S2 and combine the result
        auto tempX = 0.0f, tempY = 0.0f;
```

```
        for (auto k = 0; k < rowY; k++)
        {
            //Y-row
            for (auto l = 0; l < colY; l++)
            {
                //Y-col
                for (auto i = 0; i < rowH; i++)
                {
                    //H-row
                    for (auto j = 0; j < colH; j++)
                    {
                        //H-col
                        //-(i-k) = k+i
                        tempX += S1(i, j) * X(k + i, l + j);
                        tempY += S2(i, j) * X(k + i, l + j);
                    }
                }
                const auto res = static_cast<int>(abs(tempX) + abs(tempY));
                Y(k, l) = res;
                tempX = 0.0f, tempY = 0.0f;
            }
        }

        //write data out to file and cleanup
        pgmFile.WriteData(y, pgmFile.GetHeader());

        free(y);
        free(x);
    }

    static void Correlate(char* inFile, char* outFile, char* hFile)
    {
        PgmIO xPgm(inFile, outFile);
        PgmIO hPgm(hFile);

        auto yHeader = xPgm.GetHeader();
        const auto rowX = xPgm.GetHeader().height,
                   colX = xPgm.GetHeader().width,
                   rowH = hPgm.GetHeader().height,
                   colH = hPgm.GetHeader().width,
                   rowZ = rowX + 2 * (rowH - 1),
                   colZ = colX + 2 * (colH - 1),
                   rowY = rowX + (rowH - 1),
                   colY = colX + (colH - 1);

        yHeader.width = colY;
        yHeader.height = rowY;

        const auto x = static_cast<double*>(calloc(sizeof(double), rowZ *
colZ));
        const auto h = static_cast<double*>(calloc(sizeof(double), rowH *
colH));
        const auto x2 = static_cast<double*>(calloc(sizeof(double), rowZ *
colZ));
        const auto h2 = static_cast<double*>(calloc(sizeof(double), rowH *
colH));
```

```
        const auto y1 = static_cast<double*>(calloc(sizeof(double), rowY *
colY));
        const auto y2 = static_cast<double*>(calloc(sizeof(double), rowY *
colY));


        xPgm.GetData(x, rowH - 1, colH - 1);
        hPgm.GetData(h);

        for (auto i = 0; i < rowZ * colZ; i++)
        {
            x2[i] = pow(x[i], 2);
        }
        for (auto i = 0; i < rowH * colH; i++)
        {
            h2[i] = 1;
        }
        Convolve2D(y2, rowY, colY, x2, rowX, colX, h2, rowH, colH);
        Convolve2D(y1, rowY, colY, x, rowX, colX, h, rowH, colH);
        for (auto i = 0; i < rowY * colY; i++)
        {
            y1[i] /= y2[i];
        }
        ScaleValues(y1, rowY * colY, 255);

        xPgm.WriteData(y1, yHeader, hPgm.GetHeader().height - 1,
hPgm.GetHeader().width - 1);
        free(x);
        free(h);
        free(x2);
        free(h2);
        free(y1);
        free(y2);
    }

private:
    static void Convolve2D(double* y, const int yM, const int yN, const
double* x, const int xM, const int xN,
                           const double* h, const int hM, const int hN)
    {
#define Y(u,v) y[(u)*yN+(v)]
#define X(u,v) x[(u)*xN+(v)]
#define H(u,v) h[(u)*hN+(v)]


        //Convolve
        auto result = 0.0f;
        for (auto k = 0; k < yM; k++)
        {
            //Y-row
            for (auto l = 0; l < yN; l++)
            {
                //Y-col
                for (auto i = 0; i < hM; i++)
                {
                    //H-row
                    for (auto j = 0; j < hN; j++)
```

```
                    {
                        //H-col
                        result += H(i, j) * X(k + i, l + j);
                    }
                }
                Y(k, l) = result;
                result = 0.0f;
            }
        }
    }

    static void ScaleValues(double* y, const int n, const double maxOut)
    {
        const auto maxVal = *max_element(y, y + n);
        for (auto i = 0; i < n; i++)
        {
            y[i] = y[i] / maxVal * maxOut;
        }
    }
};
```