

Advanced Software Development – Item 1 Report

Introduction

The aim of this assignment is to implement the “Nearest Neighbour Search” algorithm for two separate tasks, and follow it up with a technical report that details how that was done.

The two tasks will focus on the subject of Computer Vision, and are as follows:

1. Create a program that restores a shuffled picture of the University of Lincoln logo
2. Create a program that detects the location of a template image in a cluttered scene

These two tasks will both be written in C++, and make use of the Matrix class that we developed within the workshops throughout the first semester.

After these tasks are done, the following questions will be researched and an answer properly concluded and discussed within the report:

1. Should operators in C++ be virtual?
2. Is a square a rectangle?

Methods (Design + Pseudocode)

matrix.h

```
class Matrix
{
private:
    int M, N;                // rows, columns
    double* data;            // pointer to actual array contents
public:
    Matrix(int, int, double); // constructor 1
    Matrix(int, int, double*); // constructor 2
    Matrix(const Matrix&);     // copy constructor
    ~Matrix();                // destructor

    int getM() const;         // returns M
    int getN() const;         // returns N
    double get(int, int) const; // get function 1
    double get(int) const;     // get function 2
    void set(int, int, double); // set function 1
    void set(int, double);     // set function 2
    void show();               // outputs the contents to console
    double* getBlock(int, int, int, int) const;

    Matrix operator+(const Matrix&); // addition operator function
    Matrix operator-(const Matrix&); // minus operator function
    Matrix operator*(const Matrix&); // multiplication operator function
    Matrix operator/(const Matrix&); // division operator function
    Matrix& operator=(const Matrix&); // equals operator function
    Matrix& operator++();             // increment operator function
    double operator()(int, int);     // "subscript" operator function
};
```

binaryimage.h

```
class BinaryImage : public Matrix
{
public:
    BinaryImage(int, int, double*, double); // constructor
    BinaryImage(const BinaryImage&);        // copy constructor
    ~BinaryImage();                         // destructor

    BinaryImage operator+(const BinaryImage&); // OR function
    BinaryImage operator-(const BinaryImage&); // XOR function
    BinaryImage operator*(const BinaryImage&); // AND function
    BinaryImage operator/(const BinaryImage&); // NAND function
    BinaryImage& operator++();                 // NOT function
};
```

main.cpp

```
double* unshuffleImage(BinaryImage&, BinaryImage&, int, int); // task 1
double* findTemplate_SAD(Matrix&, Matrix&);                  // task 2.1
double* findTemplate_SSD(Matrix&, Matrix&);                  // task 2.2
```

Task 1 – Jigsaw Solution w/ Linear Search

What this function does is take two images (the original image and the shuffled image) that are converted to BinaryImage format, which incorporates a rudimentary form of locality sensitive hashing by allowing only two values (0/1).

Each block in the original image is then compared to each block in the shuffled image, with the number of matching pixels per comparison calculated and keeping track of the one that's the best match. It then draws the best matching block to the new image data (stored in a double array, 'unshuffled').

Honestly, this is a terrible approach, but because I'm only have to compare each 32x32 block rather than every possible position, it works just fine for this task.

```
double* function unshuffleImage
{
    BinaryImage noiseImg = logo_with_noise.pgm;
    BinaryImage shuffled = logo_shuffled.pgm;

    double* unshuffled = new double[shuffled.width * shuffled.height];

    double bestValue = 255;
    int closestRow = 0;
    int closestCol = 0;

    for each 32x32 block in noiseImg
    {
        for ( int i = 0; i < 512; i += 32 )          // rows
        {
            for ( int j = 0; j < 512; j += 32 ) // columns
            {
                store number of matching pixels between
                    shuffled[i,j] and initial block;

                if value is better than bestValue
                    store new value in bestValue;
                    closestRow = i;
                    closestCol = j;
            }
        }

        add block i,j from 'shuffled' to 'unshuffled';
        bestValue = 255;
        closestRow = -1;
        closestCol = -1;
    }

    return unshuffled;
}
```

Task 2 – Template Matching w/ Linear Search + SAD and SSD

Both implementations of task 2 (findTemplate_SAD and findTemplate_SSD) use the same linear search algorithm as task 1 whilst making use of different similarity criterion – ‘sum of absolute differences’ and ‘sum of squared differences’, respectively.

```
Matrix function findTemplate_SAD
{
    Matrix image = Cluttered_scene.PGM;
    Matrix template = Wally_grey.PGM;

    Matrix result SIZE OF template;

    double bestSAD = 0;
    int x = 0;
    int y = 0;

    for every location in image
    {
        calculate SAD between template and this patch of image;

        if value is smaller than bestSAD
            bestSAD = value;
            x = current x in image;
            y = current y in image;
    }

    for each pixel in image starting at x,y
        draw pixel to result, position x,y + offset
        finish for size of template

    return result;
}
```

```
Matrix function findTemplate_SSD
{
    Matrix image = Cluttered_scene.PGM;
    Matrix template = Wally_grey.PGM;

    Matrix result SIZE OF template;

    double bestSSD = 0;
    int x = 0;
    int y = 0;

    for every location in image
    {
        calculate SSD between template and this patch of image;

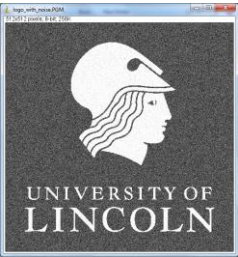



        if value is smaller than bestSSD
            bestSSD = value;
            x = current x in image;
            y = current y in image;
    }

    for each pixel in image starting at x,y
        draw pixel to result, position x,y + offset
        finish for size of template

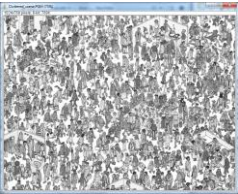

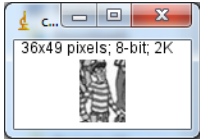
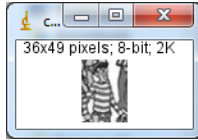
    return result;
}
```

Results

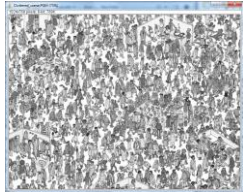

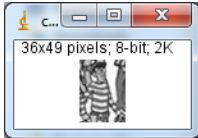

Task 1 – Jigsaw Solution w/ Linear Search

Test Description	Input Data	Expected Output	Real Output	Passed
<p>A noise filled image is loaded into the program and converted to BinaryImage format, as well as a shuffled version of the original image.</p> <p>The program must then produce an unshuffled version and store it to disk.</p>	 logo_with_noise.PGM  logo_shuffled.PGM	 <i>logo_shuffled, now completely unshuffled, stored in the same folder as:</i> unshuffledfile.pgm	 <i>logo_shuffled, now completely unshuffled, stored in the same folder as:</i> unshuffledfile.pgm	YES

Task 2.1 – Template Matching w/ Linear Search + Sum of Absolute Differences

Test Description	Input Data	Expected Output	Real Output	Passed
<p>A large image with a variety of different things on it is loaded into the program, as well as a template consisting of a small portion of that image.</p> <p>The program must then produce another image that consists of the original image cropped to the location of the template.</p>	 Cluttered_scene.PGM  Wally_grey.PGM	 <i>Cluttered_scene, cropped to the template location, stored at:</i> croppedImage.pgm	 <i>Cluttered_scene, cropped to the template location, stored at:</i> croppedImage.pgm	YES

Task 2.1 – Template Matching w/ Linear Search + Sum of Squared Differences

Test Description	Input Data	Expected Output	Real Output	Passed
<p>A large image with a variety of different things on it is loaded into the program, as well as a template consisting of a small portion of that image.</p> <p>The program must then produce another image that consists of the original image cropped to the location of the template.</p>	 <p>Cluttered_scene.PGM</p>  <p>Wally_grey.PGM</p>	 <p><i>Cluttered_scene, cropped to the template location, stored at: croppedImage_SSD.pgm</i></p>	 <p><i>Cluttered_scene, cropped to the wrong location, stored at: croppedImage_SSD.pgm</i></p>	NO

Algorithm Evaluation

The first task was solved using a linear search approach to comparing the two images, done for 256 32x32 blocks – for each of these, there's also individual comparisons for each pixel, which is 1024 iterations per block. Overall, this results in there being 262,144 pixel comparisons every time it's called – this sounds like a lot, but in reality it doesn't take that long.

However, this same approach was taken for the first solution to task 2 – the template matching – which puts it on a whole different level. Rather than comparing every 32x32 block, this function essentially slides the template image over every possible position it could be and then compares every pixel. So for the 1024x768 input, the SAD for 36x49 (1764) pixels is calculated and added together – resulting in 1,387,266,048 total operations. This takes a ridiculous amount of time.

The second implementation was the same, but used the sum of squared differences (SSD) as the similarity criterion instead. This proved to be much less accurate, whilst taking the same amount of time to process as the previous implementation.

Solution	Input Size	Total No. Operations
1	512x512 shuffled 512x512 original image 32x32 blocks	261,444
2.1	1024x768 complete image 36x49 template image	1,387,266,048
2.2		1,387,266,048

Future Work / Extensions

There are a few possible improvements that can be made to **task 1** – the `unshuffleImage()` function:

- Allowing the user to **specify the size of the blocks** themselves, rather than always assuming a 32x32 format that the example was given in.
- Improve performance by **skipping blocks that have already been checked** – for a minimal increase in memory use, the performance of the algorithm can be increased greatly as well as skipping any possibility of blocks being reused (like traditional jigsaws)
- A **visual representation in-program of the unshuffled image**, to save the user the trouble of navigating to the file themselves in their desktop environment
- Keep conversion to **BinaryImage** format within the `unshuffleImage()` function, rather than passing references to images that have already been converted – this makes the `main()` function less cluttered, and saves time when testing multiple image sets.

For **task 2** there were some obvious improvements required – while the similarity criterion I used were fine and accurate enough for the task, the speed was too slow for much practical application:

- Remove the use of linear search and replace it with a better system – possibly **fast fourier transformation**, a method of quickly computing signal correlation that has been used to great effect for template matching in certain computer vision based libraries (Itseez, 2014).

In addition to these improvements, all approaches could be improved in order to **account for there being no match found** – at the moment it will still choose the best match, even if that match still isn't anything like what it was looking for. This could be implemented by simply deciding on a threshold for importance (e.g. if the calculated number of matching pixels is less than half, flag that it didn't find a match).

Research

1. Should operators in C++ be virtual?

A virtual function is a “special kind of function that resolves to the most-derived version of the function with the same signature” (Learn C++, 2014). Methods are only made virtual then if they *require* fundamental differences in the way derived classes use them, which leaves us with no clear answer to this question – it relies entirely on the situation.

On the one hand, operator functions for a base class would work for a derived class automatically, and as such should only make adjustments to the data members that every derived class could potentially have – data that would be present in the base class already, and thus not require overloading in others.

On the other, there are certain situations in which you might want to add certain extra instructions depending on the class involved, or might even want to restrict exactly what kind of class can operate with it – as said before, if they’re all by default compatible, this might not be a desirable outcome and you might want to specify exactly what types of classes *can* operate with each other. This *would* require a virtual operator method when dealing with derived classes.

2. Is a square a rectangle?

Rectangles are parallelograms with four right angles, with opposite sides of equal length and diagonals that bisect each other. Squares are parallelograms with four right angles with *all* sides of equal length and diagonals that bisect each other.

Despite that minor difference, the rules that define a square still fall under the definition of a rectangle (if all sides are equal, then all opposite sides must be equal too), but not the other way around: all squares are rectangles, but not all rectangles are squares (Bonnie Yoder, n/a).

References

Bonnie Yoder (n/a) ***Why is a Square also a Rectangle*** [online]
Available from: <https://www.math.okstate.edu/geoset/Projects/Ideas/SquareRect.htm>
[Accessed 24th January 2014]

Learn C++ (2014) **12.2 – Virtual functions « Learn C++** [online]
Available from: <http://www.learncpp.com/cpp-tutorial/122-virtual-functions/>
[Accessed 24th January 2014]

Appendix – Source Code

matrix.h

```
// matrix.h : data member and function definitions for the Matrix class

#ifndef _matrixclass          // if the Matrix class hasn't already been defined...
#define _matrixclass         // ...flag that it has been

class Matrix
{
private:
    int M, N;                // rows, columns
    double* data;            // pointer to actual array contents
public:
    Matrix(int, int, double); // constructor 1
    Matrix(int, int, double*); // constructor 2
    Matrix(const Matrix&);     // copy constructor
    ~Matrix();                // destructor

    int getM() const;         // returns M
    int getN() const;         // returns N
    double* getData() const;  // returns pointer to data

    double get(int, int) const; // get function 1
    double get(int) const;      // get function 2
    void set(int, int, double); // set function 1
    void set(int, double);      // set function 2
    void show();               // outputs the contents to console

    double* getBlock(int, int, int, int) const;

    Matrix operator+(const Matrix&); // addition operator function
    Matrix operator-(const Matrix&); // minus operator function
    Matrix operator*(const Matrix&); // multiplication operator function
    Matrix operator/(const Matrix&); // division operator function

    Matrix& operator=(const Matrix&); // equals operator function
    Matrix& operator++();              // increment operator function
    double operator()(int, int);      // "subscript" operator function
};

#endif
```

binaryimage.h

```
// binaryimage.h : derived class of matrix, allows only binary values (0/1)
#include "matrix.h"

#ifndef _binaryimage
#define _binaryimage

class BinaryImage : public Matrix
{
public:
    BinaryImage(int, int, double*, double);           // constructor
    BinaryImage(const BinaryImage&);                 // copy constructor
    ~BinaryImage();                                   // destructor

    BinaryImage operator+(const BinaryImage&);       // OR function
    BinaryImage operator-(const BinaryImage&);       // XOR function
    BinaryImage operator*(const BinaryImage&);       // AND function
    BinaryImage operator/(const BinaryImage&);       // NAND function

    BinaryImage& operator++();                        // NOT function
};

#endif
```

imagemethods.h

```
// imagemethods.h : methods for reading and writing .PGM files

#ifndef _imagemethods
#define _imagemethods

double* readImage(char* file, int r, int c);
void writeImageToFile(char* file, double* data, int r, int c, int q);

#endif
```

main.cpp

```
#include <iostream>
#include <math.h>
#include "matrix.h"
#include "binaryimage.h"
#include "imagemethods.h"

using namespace std;

double* unshuffleImage(BinaryImage&, BinaryImage&, int, int);
double* findTemplate_SAD(Matrix&, Matrix&);
double* findTemplate_SSD(Matrix&, Matrix&);

int main()
{
    ////////////////////////////////////////////////////
    // TASK 1 - Unshuffled Logo    //
    ////////////////////////////////////////////////////

    // setup for task 1
    cout << "Loading images for task 1..." << endl;
    double* noisyImage = readImage("Task 1/logo_with_noise.txt", 512, 512);
    double* shuffledImage = readImage("Task 1/logo_shuffled.txt", 512, 512);

    // convert images to binary (locality sensitive hashing)
    cout << "Converting images to binary format..." << endl;
    BinaryImage noisyImageBin(512, 512, noisyImage, 150);
    BinaryImage shuffledImageBin(512, 512, shuffledImage, 200);

    // unshuffle image and write to file
    cout << "Unshuffling logo..." << endl;
    double* result_task1 = unshuffleImage(noisyImageBin, shuffledImageBin,
                                          512, 512);
    writeImageToFile("Task 1/unshuffled.PGM", result_task1, 512, 512, 1);
    cout << "Logo saved to 'Task 1/unshuffled.PGM'" << endl;

    // cleanup for task 1
    cout << "Cleaning up task 1 memory..." << endl;
    delete[] noisyImage; noisyImage = 0;
    delete[] shuffledImage; shuffledImage = 0;
    delete[] result_task1; result_task1 = 0;

    ////////////////////////////////////////////////////
    // TASK 2 - Template Matching  //
    ////////////////////////////////////////////////////

    // setup for task 2
    cout << "Loading images for task 2..." << endl;
    double* imageHolder = readImage("Task 2/Cluttered_scene.txt", 768, 1024);
    double* templateHolder = readImage("Task 2/Wally_grey.txt", 49, 36);

    // convert data to Matrix format
    Matrix image(768, 1024, imageHolder);
    Matrix templateImage(49, 36, templateHolder);

    // task 2.1 - sum of absolute differences
    cout << "Running SAD implementation of task 2 (this can take a while)..."
        << endl;
    double* cropped = findTemplate_SAD(image, templateImage);
```

```

writeImageToFile("Task 2/croppedImage_SAD.pgm", cropped, 49, 36, 255);
cout << "Result saved to 'Task 2/croppedImage_SAD.PGM'" << endl;

// task 2.2 - sum of squared differences
cout << "Running SSD implementation of task 2 (this can take a while)..." <<
endl;
double* cropped2 = findTemplate_SSD(image, templateImage);
writeImageToFile("Task 2/croppedImage_SSD.pgm", cropped2, 49, 36, 255);
cout << "Result saved to 'Task 2/croppedImage_SSD.PGM'" << endl;

// clean up memory
cout << "Cleaning up task 2 memory..." << endl;
delete[] imageHolder; imageHolder = 0;
delete[] templateHolder; templateHolder = 0;
delete[] cropped; cropped = 0;
delete[] cropped2; cropped2 = 0;

// finish program, return standard exit integer (0)
system("pause");
return 0;
}

double* unshuffleImage(BinaryImage& image, BinaryImage& shuffled, int width, int
height)
{
    // create new array of doubles to hold the output
    double* unshuffled = new double[width * height];

    int bestValue = 0, holderValue = 0;
    int row = 0, col = 0;

    // look through each block of the original image
    for ( int i = 0; i < height; i+=32 )
    {
        for ( int j = 0; j < width; j+=32 )
        {
            // and compare against each block in the shuffled image
            for ( int i2 = 0; i2 < height; i2+=32 )
            {
                for ( int j2 = 0; j2 < width; j2+=32 )
                {
                    // and within that, calculate how many of the
                    pixels match...
                    for ( int a = 0; a < 32; a++ )
                    {
                        for ( int b = 0; b < 32; b++ )
                        {
                            if ( a > 5 && a < 27 && b > 5 && b <
27 )
                                continue;

                            if ( image(i+a, j+b) == shuffled(i2+a,
j2+b) )
                                holderValue += 1;
                        }
                    }

                    // if it's better than the previous best...
                    if ( holderValue > bestValue )
                    {
                        // store the appropriate values
                        bestValue = holderValue;
                    }
                }
            }
        }
    }
}

```

```
        row = i2; col = j2;
    }

    holderValue = 0;
}

// store the correct block details in 'unshuffled'
for ( int a = 0; a < 32; a++ )
{
    for ( int b = 0; b < 32; b++ )
    {
        unshuffled[((i+a)*width) + (j + b)] =
shuffled(row+a, col+b);
    }
}

// reset values for later
bestValue = 0; holderValue = 0;
row = 0; col = 0;
}

// return pointer to caller
return unshuffled;
}

double* findTemplate_SAD(Matrix& image, Matrix& tempImage)
{
    double* finalImage = new double[tempImage.getM() * tempImage.getN()];

    double bestSAD = 0, SAD = 0;
    int x = 0, y = 0;

    // for each part of the original image...
    for ( int i = 0; i < image.getM() - tempImage.getM(); i ++ )
    {
        for ( int j = 0; j < image.getN() - tempImage.getN(); j ++ )
        {
            // calculates the SAD value between each of the pixels
            for ( int i2 = 0; i2 < tempImage.getM(); i2++ )
            {
                for ( int j2 = 0; j2 < tempImage.getN(); j2++ )
                {
                    SAD += abs(image(i+i2,j+j2) - tempImage(i2,j2));
                }
            }

            // if this value is better than the previous best, store values
            if ( SAD < bestSAD || (i == 0 && j == 0))
            {
                bestSAD = SAD;
                x = j; y = i;
            }

            // resets the SAD value holder for the next attempt
            SAD = 0;
        }
    }

    // draw the correct part of the original image to finalImage
    for ( int i = y; i < y + tempImage.getM(); i++ )
```

```

    {
        if ( i >= image.getM() ) break;
        for ( int j = x; j < x + tempImage.getN(); j++ )
        {
            if ( j >= image.getN() ) break;
            finalImage[((i-y)*tempImage.getN()) + (j-x)] = image(i,j);
        }
    }

    // return the result to the caller as a pointer
    return finalImage;
}

double* findTemplate_SSD(Matrix& image, Matrix& tempImage)
{
    double* finalImage = new double[tempImage.getM() * tempImage.getN()];

    double bestSSD = 0, SSD = 0;
    int x = 0, y = 0;

    // for each part of the original image...
    for ( int i = 0; i < image.getM() - tempImage.getM(); i ++ )
    {
        for ( int j = 0; j < image.getN() - tempImage.getN(); j ++ )
        {
            // calculates the SSD value between each of the pixels
            for ( int i2 = 0; i2 < tempImage.getM(); i2++ )
            {
                for ( int j2 = 0; j2 < tempImage.getN(); j2++ )
                {
                    SSD += (image(i+i2,j+j2) - tempImage(i2,j2)) *
(image(i+i2,j+j2) - tempImage(i2,j2));
                }
            }

            // if this value is better than the previous best, store values
            if ( SSD < bestSSD || (i == 0 && j == 0))
            {
                bestSSD = SSD;
                x = j; y = i;
            }

            // resets the SSD value holder for the next attempt
            SSD = 0;
        }
    }

    // draw the correct part of the original image to finalImage
    for ( int i = y; i < y + tempImage.getM(); i++ )
    {
        if ( i >= image.getM() ) break;
        for ( int j = x; j < x + tempImage.getN(); j++ )
        {
            if ( j >= image.getN() ) break;
            finalImage[((i-y)*tempImage.getN()) + (j-x)] = image(i,j);
        }
    }

    // return the result to the caller as a pointer
    return finalImage;
}

```

matrix.cpp

```
#include <iostream>
#include "matrix.h"

using namespace std;

////////////////////
// Matrix : IMPLEMENTATION //
////////////////////

// constructor 1 : initializes rows, columns, and sets value across all addresses
Matrix::Matrix(int sizeR, int sizeC, double val)
{
    M = sizeR;                                // sets number of rows
    N = sizeC;                                // sets number of
columns
    data = new double[M * N];                 // sets aside right amount of space in
heap for data

    for ( int i = 0; i < M * N; i++ ) // traverses array's memory addresses
    {
        *(data+i) = val;                  // initializes each address
with the passed value ('val')
    }

    // output message to the console
    cout << "Matrix::Matrix(int, int, double) is invoked..." << endl;
}

// constructor 2 : initializes rows, columns, and copies over data values from a
previous Matrix
Matrix::Matrix(int sizeR, int sizeC, double* input_data)
{
    M = sizeR;                                // sets number of rows
    N = sizeC;                                // sets number of
columns
    data = new double[M * N];                 // sets aside right amount of space in
heap for data

    for ( int i = 0; i < M * N; i++ ) // traverses array
    {
        *(data + i) = *(input_data + i); // copies value over to new Matrix
    }

    // output message to the console
    cout << "Matrix::Matrix(int, int, double*) is invoked..." << endl;
}

// copy constructor
Matrix::Matrix(const Matrix& mA)
{
    // sets the boundaries of this matrix to match mA's
    M = mA.getM(); N = mA.getN();

    // set aside the right amount of memory
    data = new double[M * N];

    // populates the data values (deep copy)
    for ( int i = 0; i < M * N; i ++ )
```

```
        *(data + i) = mA.get(i);

        // output message to the console
        cout << "Matrix::Matrix(const Matrix&) is invoked..." << endl;
    }

    // destructor
    Matrix::~Matrix()
    {
        // clear all the data within the Matrix
        delete[] data; data = 0;

        // output message to the screen
        cout << "Matrix::~Matrix() is invoked..." << endl;
    }

    // int get_() : returns either M or N, depending on the function called
    int Matrix::getM() const
    {
        return M;
    }
    int Matrix::getN() const
    {
        return N;
    }
    double* Matrix::getData() const
    {
        return data;
    }

    // double const get(int, int) : returns specific value from a selected point in the
    // array
    double Matrix::get(int i, int j) const
    {
        return *(data + (i * N) + j);
    }

    // double const get(int, int) : returns specific value from a selected point in the
    // array
    double Matrix::get(int i) const
    {
        return *(data + i);
    }

    // void set(int, int, double) : sets specific value at a selected point in the array
    void Matrix::set(int i, int j, double val)
    {
        *(data + (i * N) + j) = val;
    }

    // void set(int, double) : sets specific value at selected point
    void Matrix::set(int i, double val)
    {
        data[i] = val;
    }

    // void show() : outputs the contents of the array to the console
    void Matrix::show()
    {
        for ( int i = 0; i < M; i++ )           // traverse rows in array
        {
            for ( int j = 0; j < N; j++ )       // traverse columns in array
```



```

        {
            //cout << *(data + (i*N) + j) << "\t";
            cout << get(i,j) << "\t";
        }
        cout << endl;
    }
}

// double* getBlock() : returns a pointer to a copied sub-array of the Matrix's array
double* Matrix::getBlock(int start_row, int end_row, int start_col, int end_col) const
{
    // store the number of rows and columns in the subarray
    int newM = (end_row - start_row) + 1;
    int newN = (end_col - start_col) + 1;

    // set aside the right amount of memory
    double* newData = new double[newM * newN];

    // populate the array with the correct values
    for ( int i = 0; i < newM; i++ )          // traverse rows
    {
        for ( int j = 0; j < newN; j++ )      // traverse columns
        {
            *(newData + (i * newN) + j) = get(start_row + i, start_col + j);
        }
    }

    // return pointer to the array
    return newData;
}

// operator functions
Matrix Matrix::operator+(const Matrix& B)
{
    Matrix C(M, N, 0.0);          // create new matrix

    for ( int i = 0; i < M && i < B.getM(); i++ )    // traverse rows
    {
        for ( int j = 0; j < N && j < B.getN(); j++ )    // traverse columns
        {
            // add the two values together and store in C
            C.set(i, j, (*this).get(i, j) + B.get(i, j));
        }
    }

    // output message to the console
    cout << "Matrix::operator+() was invoked...";
    // return new matrix
    return C;
}

Matrix Matrix::operator-(const Matrix& B)
{
    Matrix C(M, N, 0.0);          // create new matrix

    for ( int i = 0; i < M && i < B.getM(); i++ )    // traverse rows
    {
        for ( int j = 0; j < N && j < B.getN(); j++ )    // traverse columns
        {
            // add the two values together and store in C
            C.set(i, j, (*this).get(i, j) - B.get(i, j));
        }
    }
}

```

```
// output message to the console
cout << "Matrix::operator-() was invoked...";
// return new matrix
return C;
}
Matrix Matrix::operator*(const Matrix& B)
{
    Matrix C(M, N, 0.0);          // create new matrix

    for ( int i = 0; i < M && i < B.getM(); i++ )    // traverse rows
    {
        for ( int j = 0; j < N && j < B.getN(); j++ )    // traverse columns
        {
            // add the two values together and store in C
            C.set(i, j, (*this).get(i, j) * B.get(i, j));
        }
    }

    // output message to the console
    cout << "Matrix::operator*() was invoked...";
    // return new matrix
    return C;
}
Matrix Matrix::operator/(const Matrix& B)
{
    Matrix C(M, N, 0.0);          // create new matrix

    for ( int i = 0; i < M && i < B.getM(); i++ )    // traverse rows
    {
        for ( int j = 0; j < N && j < B.getN(); j++ )    // traverse columns
        {
            // add the two values together and store in C
            C.set(i, j, (*this).get(i, j) / B.get(i, j));
        }
    }

    // output message to the console
    cout << "Matrix::operator/() was invoked...";
    // return new matrix
    return C;
}
Matrix& Matrix::operator=(const Matrix& B)
{
    // output message to the console
    cout << "Matrix::operator=() is invoked...";

    // check to see they're not the same
    if ( this == &B ) return *this;
    // continue on otherwise...
    else
    {
        // delete old data
        delete[] data; data = 0;

        // assign new row and column values
        M = B.getM(); N = B.getN();

        // allocate memory
        data = new double[M * N];
    }
}
```

```
        // populate values of the array
        for ( int i = 0; i < M * N; i++ )
            *(data + i) = B.get(i);

        // return matrix
        return *this;
    }
}

Matrix& Matrix::operator++()
{
    // increment every data value held in the array
    for ( int i = 0; i < M * N; i++ )
        *(data + i) = *(data + i) + 1;

    // output message to the console
    cout << "Matrix::operator++() is invoked...";

    // return matrix
    return *this;
}

double Matrix::operator()(int i, int j)
{
    // output message to the console
    //cout << "Matrix::operator() is invoked...";

    // return value to the caller
    return *(data + (i * N) + j);
}
```

binaryimage.cpp

```
#include <iostream>
#include "binaryimage.h"

using namespace std;

////////////////////////////////////
// BinaryImage : IMPLEMENTATION //
////////////////////////////////////

// default constructor
BinaryImage::BinaryImage(int M, int N, double* input_data, double thresh) : Matrix(M,
N, input_data)
{
    for ( int i = 0; i < M; i++ )           // traverse rows
    {
        for ( int j = 0; j < N; j++ )       // traverse columns
        {
            // sets to 1 or 0 based on the threshold value
            if ( get(i, j) > thresh ) set(i, j, 1);
            else set(i, j, 0);
        }
    }

    // output message to the console
    cout << "BinaryImage::BinaryImage(int, int, double*, double) is invoked..." <<
endl;
}

// copy constructor
BinaryImage::BinaryImage(const BinaryImage& biB):Matrix(biB)
{
    // output message to the console
    cout << "BinaryImage::BinaryImage(const BinaryImage&) was invoked..." << endl;
}

// destructor
BinaryImage::~BinaryImage()
{
    // output message to the console
    cout << "BinaryImage::~BinaryImage() is invoked..." << endl;
}

// operator+() / OR function
BinaryImage BinaryImage::operator+(const BinaryImage& biB)
{
    // creates new binary image with same boundaries as this
    BinaryImage C = *this;

    // populate the values held within the image
    for ( int i = 0; i < getM() && i < biB.getM(); i++ )           // traverse rows
    {
        for ( int j = 0; j < getM() && j < biB.getM(); j++ )       // traverse
columns
        {
            // if either has the value '1', then assign value '1', else '0'
            C.set(i, j, (get(i,j) == 1 || biB.get(i,j) == 1) ? 1 : 0 );
        }
    }
}
```

```
        // returns new binary image
        return C;
    }

    // operator-() / XOR function
    BinaryImage BinaryImage::operator-(const BinaryImage& biB)
    {
        // creates new binary image with same boundaries as this
        BinaryImage C = *this;

        // populate the values held within the image
        for ( int i = 0; i < getM() && i < biB.getM(); i++ )           // traverse rows
        {
            for ( int j = 0; j < getN() && j < biB.getN(); j++ )       // traverse
columns
            {
                // if ONLY a single one has the value '1', then assign value '1',
else '0'
                C.set(i, j, ((get(i,j) == 1 && biB.get(i,j) == 0) || (get(i,j) ==
0 && biB.get(i,j) == 1)) ? 1 : 0 );
            }
        }

        // returns new binary image
        return C;
    }

    // operator*() / AND function
    BinaryImage BinaryImage::operator*(const BinaryImage& biB)
    {
        // creates new binary image with same boundaries as this
        BinaryImage C = *this;

        // populate the values held within the image
        for ( int i = 0; i < getM() && i < biB.getM(); i++ )           // traverse rows
        {
            for ( int j = 0; j < getN() && j < biB.getN(); j++ )       // traverse
columns
            {
                // if both have the value '1', then assign '1', else '0'
                C.set(i, j, (get(i,j) == 1 && biB.get(i,j) == 1) ? 1 : 0 );
            }
        }

        // returns new binary image
        return C;
    }

    // operator/() / NOT AND function
    BinaryImage BinaryImage::operator/(const BinaryImage& biB)
    {
        // creates new binary image with same boundaries as this
        BinaryImage C = *this;

        // populate the values held within the image
        for ( int i = 0; i < getM() && i < biB.getM(); i++ )           // traverse rows
        {
            for ( int j = 0; j < getN() && j < biB.getN(); j++ )       // traverse
columns
            {
                // if ONLY a single one has the value '1', then assign value '0',
```

```
else '1'
    C.set(i, j, ((get(i,j) == 1 && biB.get(i,j) == 0) || (get(i,j) ==
0 && biB.get(i,j) == 1)) ? 0 : 1 );
    }

    // returns new binary image
    return C;
}

// operator++() / XOR function
BinaryImage& BinaryImage::operator++()
{
    // populate the values held within the image
    for ( int i = 0; i < getM(); i++ )          // traverse rows
    {
        for ( int j = 0; j < getN(); j++ ) // traverse columns
        {
            // flips every value from 1 to 0, and vice versa
            set(i, j, get(i, j) == 0 ? 1 : 0);
        }
    }

    // returns binary image
    return *this;
}
```

imagemethods.cpp

```
#include <sstream>           // stringstream
#include <iostream>          // cout, cerr
#include <fstream>           // ifstream
#include <istream>

using namespace std;

// reads in a .txt file representing an image, name 'fileName', size rows x cols
// converts to a 1D array of doubles
double* readImage(char* fileName, int rows, int cols)
{
    // sets aside the memory for the image in heap
    double* imageArray = new double[rows * cols];

    // opens the file
    ifstream txtFile(fileName);

    // if the file was successfully opened...
    if ( txtFile.is_open() )
    {
        // counter, used for pointing to correct address in imageArray
        int i = 0;

        // while there's still more to be read from the file...
        while ( !txtFile.eof() )    // (/.good())
        {
            // if there's no more space in the array, stop reading file
            if ( i >= rows * cols ) break;

            // put the next value into the array, and increment 'i'
            txtFile >> *(imageArray + i++);
        }

        // close the file
        txtFile.close();
    }
    // otherwise output an error message to the console
    else cout << "ERROR: unable to open file for reading." << endl;

    // returns pointer to data
    return imageArray;
}

// takes an array of doubles in heap memory, converts it into a .PGM file
void writeImageToFile(char* fileName, double* data, int rows, int cols, int q)
{
    // sets aside the right amount of memory in heap for converting the doubles to
    // chars
    unsigned char* newImage = new unsigned char[rows * cols];

    // actually converts the doubles in 'data' to unsigned chars in 'newImage'
    // note: unsigned chars necessary, since they have value of 0 to 255, not -128
    // to 127
    for ( int i = 0; i < rows * cols; i++ )
        newImage[i] = (unsigned char)data[i];

    // opens the file location for writing to (consider moving declaration to top)
    ofstream newFile;
    newFile.open(fileName, ios::out|ios::binary|ios::trunc);
}
```

```
// if the file was unable to be written...
if ( !newFile )
{
    // outputs an error message to the console
    cout << "ERROR: unable to open file for writing." << endl;

    // exits out of process and cleans up memory
    exit(1);
}

newFile << "P5" << endl;                                // specifies file
format
newFile << cols << " " << rows << endl;                // specifies image width
and height
newFile << q << endl;                                    //
specifies maximum pixel value (max = white, 0 = black)

// writes the rest of the image data into the file
newFile.write((char*)newImage, (rows * cols) * sizeof(unsigned char));

// if there's a problem with writing the file...
if ( newFile.fail() )
{
    // output a message to the console
    cout << "ERROR: Unable to write image to PGM file." << endl;
    exit(0);
}

// close file
newFile.close();

// delete image data from heap
delete[] newImage; newImage = 0;
}
```