CS103L PA3 – It's So Belurry

1 Introduction

In this assignment you will design and implement a program to perform simple kernel based image processing filters on an image. We will be implementing three filters: 1) the Sobel operator for edge detection (using a fixed sized kernel), 2) the Gaussian blur filter for low-pass filtering, and 3) unsharp mask filter for sharpening an image (which uses the Gaussian blur filter). The kernel-based implementation you will design forms the basis for a large-class of image processing techniques, therefore through this programming assignment you will learn to implement an algorithm with wide applicability across computer science.

Tommy Trojan filtered with a Gaussian blur filter

Black and White tiles filtered with the Sobel Kernel

USC v. UCLA filtered with the unsharp mask operation

2 What you will learn

This assignment will expose you to simple image representation and manipulation techniques as well as familiarize you with C/C++ operations on 2D arrays.

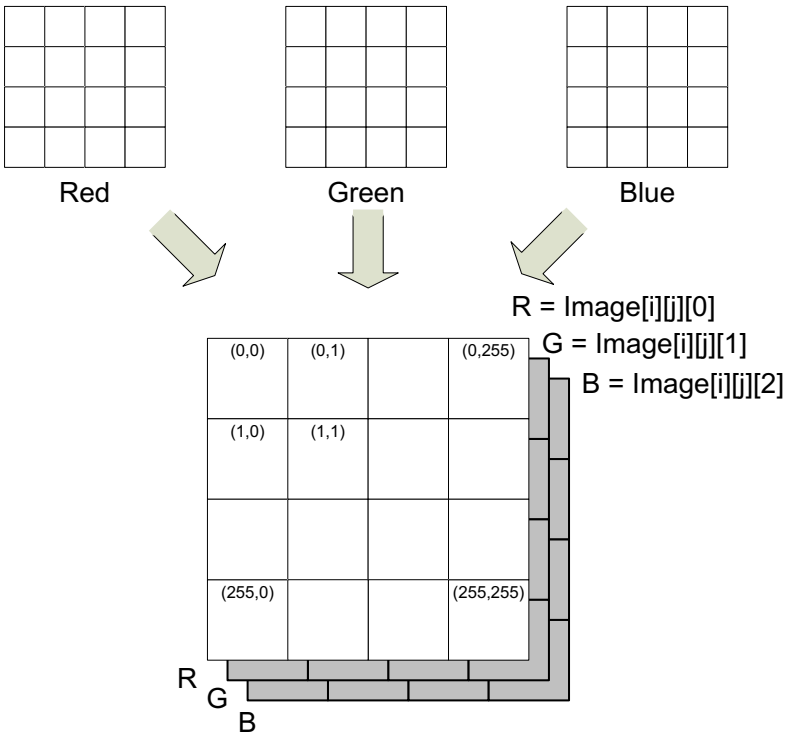
1. Understand multiple file compilation units and their linkage.
2. Use command line arguments to provide input to the program vs. interactive user input
3. Understand image representation as a collection of pixels and understand the RGB color space
4. Apply knowledge of arrays (including multi-dimensional arrays) to implement an image-processing application
5. Create, develop, and evaluate your own processing approach to perform the operation

3 Color Images as 3D Arrays in C/C++

Image processing is a major subfield of computer science and electrical engineering and to a lesser extent biomedical engineering. Graphics are usually represented via two methods: vector or bitmap. Vector graphics take a more abstract approach and use mathematical equations to represent lines, curves, polygons and their fill color. When a program opens the vector image it has to translate those equations and render the image. This vector approach is often used to represent clipart and 3D animations. Bitmap images take the opposite approach and simply represent the image as a 2D array of pixels. Each pixel is a small dot or square of color. The bitmap approach is used most commonly for pictures, video, and other images. In addition, bitmaps do not force us to translate the vector equations, and thus are simpler to manipulate for our purposes.

Color bitmaps can use one of several different color representations. The simplest of these methods is probably the RGB color space (HSL, HSV, & CMYK are others), where each pixel has separate red, green, and blue components that are combined to produce the desired color. Usually each component is an 8-bit (`unsigned char` in C) value. Given 3-values this yields a total of 24-bits for representing a specific color (known as 24-bit color = 2^{24} = 16 million unique colors). Storing a red, green and blue value for each pixel can be achieved using 3 separate 2D arrays (one for each RGB component) or can be combined into a 3D array with dimensions [256][256][3] as shown below.

[Note: BMP image files use a simple format like this one and thus will be the file format used in our lab.]



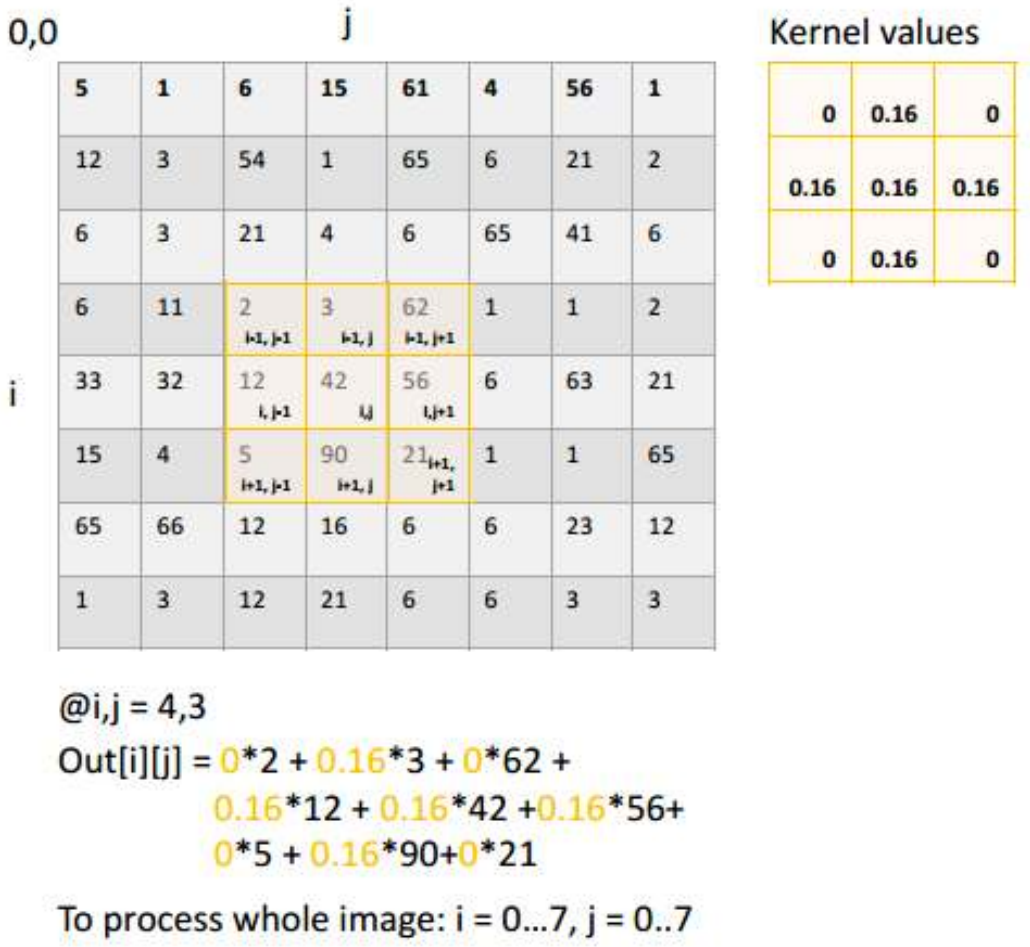
Color	RGB Value
White	255,255,255
Red	255,0,0
Yellow	255,255,0
Orange	255,128,0
Blue	0,0,255
Green	0,255,0
Purple	255,0,255
Black	0,0,0

Figure 1 – Representation of a color bitmap as a 3D array

4 Kernel Based Image Processing Algorithms

Similar to how images are represented in a computer, the most common approach to image processing techniques also uses a matrix representation. Kernel based image processing algorithms define a small matrix (a.k.a. the kernel) of floating-point numbers, which is then convolved with the source image. Convolution is a common mathematical technique, and in the case of images is not difficult to implement. To do the convolution the center of the kernel is aligned with each pixel in the source image. At each pixel the floating-point numbers in the kernel represent how much of each source image pixel contribute to the pixel in the output image.

The figure below shows an example for a 3x3 kernel known as the “cross” kernel. For each (i,j) pixel in the input image we calculate an output pixel using the 3x3 kernel. You can think of starting with the center of the kernel over pixel (0,0) and sliding it over the rest of the image one pixel at a time.



Based on this example for a grayscale image (or one color plane of a color image) we can write an equation that represents the pixel value we need to calculate at position (x,y) given a 256x256 input image (In), a NxN sized kernel (K) and a 256x256 output image (Out):

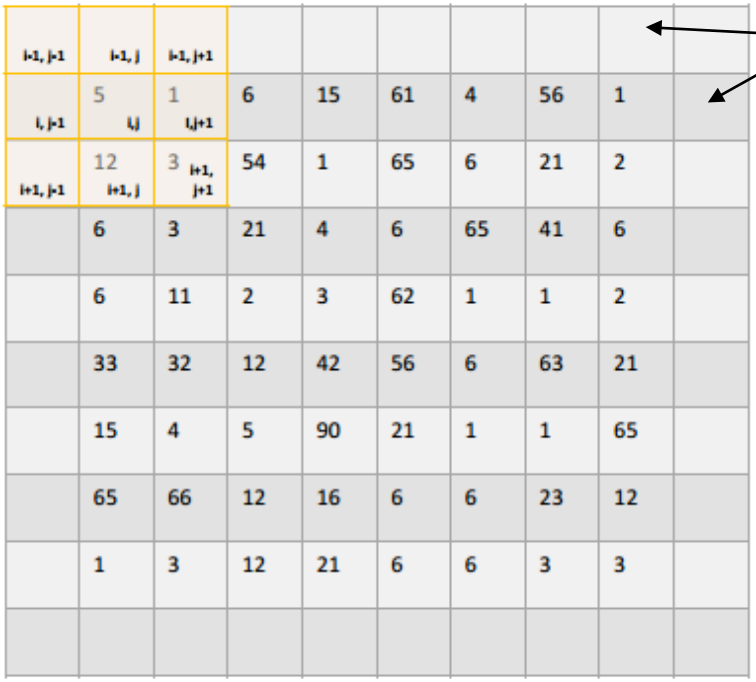
$$Out[y][x] = \sum_{i=-\frac{N}{2}}^{\frac{N}{2}} \sum_{j=-\frac{N}{2}}^{\frac{N}{2}} In[y+i][x+j] * K[\frac{N}{2}+i][\frac{N}{2}+j]$$

One issue is that our output pixel value due to this operations may produce a result outside of the range 0-255 that unsigned char's can support. Thus we should store our result in an integer, then check if it lies outside of the range 0-255. If it is less than 0, just set it to 0. If it

is greater than 255, just set it to 255. This is known as **clamping**. As a rule of thumb you should store the results of your convolution as temporary **int** types and then perform clamping to produce the actual pixel value you place in the output array.

Another issue with the equation above is that it only holds for pixels away from the edge of the image. When x < N/2 or y < N/2, the indexes will have negative values. Similarly, when x > 255 – (N/2) or y > 255 – (N/2) the indexes will have values greater than 255. These are what are called edge cases and can be handled in several ways. In this programming assignment you will copy the input image into an array that is slightly bigger than the original image. The extra pixels will be filled with a value and then the convolution is started at the offset original image. This technique is called padding and is straightforward to implement. The next figure illustrates this technique.

We see our original image surrounded by 1 pixel of padding. This will allow us to use the equation above (with careful selection of the starting and ending values of i and j). What goes in the padding is up to the programmer however, for this lab we will pad with zeros (i.e. black pixels). **You will need to pad with N/2 rows/columns to support an NxN kernel.**



Pad with 0's around the border. This would work for a 3x3 kernel. How much padding would be needed for an 11x11 kernel?

5 The Sobel Operator

For our first image processing filter we will implement a filter defined by a fixed kernel size. The Sobel implementation we will do has two kernels, horizontal 1 and horizontal 2:

Horizontal 1 Sobel Kernel

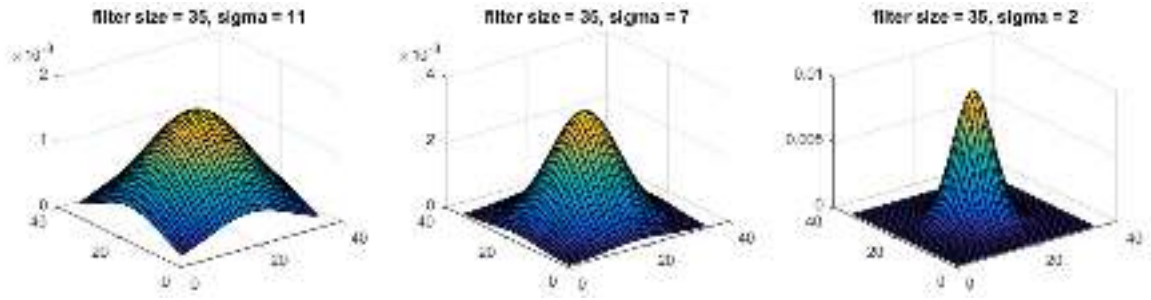
-1	0	1
-2	0	2
-1	0	1

Horizontal 2 Sobel Kernel

1	0	-1
2	0	-2
1	0	-1

This simple kernel was developed to detect (or highlight) edges in an image. The kernels can be used alone or in combination. To use them in combination we can simply apply each one to the input and produce 2 separate arrays/images. Then add the separate arrays/images together (clamping at 0 and 255) to produce the final output.

6 The 2D Gaussian Filter



The 2D Gaussian distribution is a well-known function of two variables that finds application in probability and statistics, chemistry, quantum mechanics and of course image processing. It has the following form:

$$g(x,y) = A * e^{-\left(\frac{(x-x_0)^2}{2\sigma_x^2}+\frac{(y-y_0)^2}{2\sigma_y^2}\right)}$$

Where (x_0,y_0) is the center position, *A* is the amplitude, σ is the variance. The figure above shows three 2D Gaussians with several different values of σ.

For image processing as a blur filter, the wider the central peak, the greater the blurring effect. Our task then is to take the equation above and generate the NxN kernel needed to filter an image. To do so we set A=1, and set the center point of the kernel as x_0=0, y_0=0. The variance, σ , is a parameter that can be used to adjust the size of the central peak. The x,y values of the other kernel cells are set as offsets from the center. For example, a 3x3 kernel is shown below.

Then for each cell, the Gaussian equation is evaluated and the floating point value assigned. Finally, we must normalize the values so that the sum of the values is equal to 1. This is done so that the brightness of the image does not change. The two tables below show the raw and normalized values for the 3x3 Gaussian blur kernel (N=3, sigma=1.5).

Raw 3x3 Gaussian

0.6412	0.8007	0.6412
0.8007	1	0.8007
0.6412	0.8007	0.6412

Normalized 3x3 Gaussian

0.0947	.1183	.0947
0.1183	.1478	0.1183
0.0947	0.1183	0.0947

One issue to be aware of when you produce the Gaussian kernel is indexing.
To correctly apply the Gaussion equation (0,0) should be the center of the kernel. However, for array indexing 0,0 is always the upper left. Think about how you can convert the array indexing to generate the appropriate Guassian indices.

Gaussian 3x3 row/column indexing perspective

-1,-1	-1,0	-1,1
0,-1	0,0	0,1
1,-1	1,0	1,1

C++ 2D array indexing perspective

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

7 The Unsharp-mask filter

Some image processing filters can be quickly constructed from adding, subtracting and scaling the results of other filters. The unsharp-mask filter is one such filter.

Sharpening an image is the opposite of blurring and image. Sharpening an image attempts to enhance detail. If you subtract a blurred version of an image from the original image, intuitively the resulting image will have the most detail remaining in areas that had a lot of

detail in the original image. Thinking about it another way, in areas of the original image with high-detail the blur filter will 'do more' than in areas with low detail. If we take this 'detail' map and add it back to the original image, we will enhance areas with a lot of detail, in effect sharpening the image.

If we have an original image IM we can create a blurred version B by applying the Gaussian blur function blur():

$$B = blur(IM)$$

We can then create a detail map D by subtracting B from IM. Note: The equation below is a matrix operation and implies we perform the subtraction on every corresponding pixel in the image (i.e. D[0][0] = IM[0][0] – B[0][0], D[0][1] = IM[0][1] – B[0][1], etc.)

$$D = IM - B$$

Finally, we can create the enhanced, or sharpened image S by adding some fraction of D back to IM (again this is a matrix operation where we perform the following on each pixel):

$$S = IM + \alpha D$$

Where α typically ranges from 0.3-0.7

8 Dummy Kernel

The following kernel will simply copy an input image to an output image. It may be useful for testing purposes.

0.0	**0.0**	**0.0**
0.0	**1.0**	**0.0**
0.0	**0.0**	**0.0**

9 C/C++ Language Details

This section contains helpful background information. You should read through it to understand various aspects of the C++ language. It is meant just as a review and overview of related concepts.

Some features of C that we will utilize include:

2- and 3-D Arrays: We will store the mask and images in 2- and 3-D arrays, respectively. Recall the declaration and access syntax:

```
int example2d[10][5];
unsigned char array3d[256][256][3];

x = example2d[9][4];    // accesses element in the bottom row, last column
pixel_red = array3d[255][255][0];  // access lower, right pixel's red value
```

Note that each of the RED, GREEN, and BLUE components of a pixel is an 8-bit value, logically between 0 and 255. Thus we will use the `unsigned char` type to represent these numbers.

Also, the library functions that we will provide use the convention that the **first index is the row** and the **second index is the column** (with the **third index being the R,G, or B value**.)

Math functions: You will need to compute exponents and also use the constant *e*. C++ provides an exponential function in the <cmath> library. The function calculates the value of e^x for some double *x*. The function is prototyped as:

```
double exp(double);
```

BMP Library & File I/O: For this lab, we will provide you predefined functions that you can use to read in and write out .bmp image files. These functions are given in the compiled library `bmplib.o` and are prototyped in the header file `bmplib.h`. Be sure to include `bmplib.h` in your program by adding the following line with the other #include statements.

```
#include "bmplib.h"
```

The functions you will use are "readRGBBMP" and "writeRGBBMP". Prototypes are shown below. You must pass each function a character array (text string) of the filename you wish to read/write and a 256x256x3 array of unsigned char's (8-bit values) that represent the data to be read/written.

```
int readRGBBMP(char filename[], unsigned char inputImage[][256][3]);
int writeRGBBMP(char filename[], unsigned char outputImage[][256][3]);
```

Note: These functions return 0 if successful and non-zero if they cannot open, read, or write the particular file.

For debugging purposes, you may also use the function
`void showRGBBMP(unsigned char outputImage[][256][3]);`
that displays an image. However, it may not work if you are not using the VM.

Command Line Arguments: Rather than prompting the user during your program to enter the files you will process, we will pass the filenames to your program as command line arguments. Command line arguments provide a way to pass a program some initial input values without having to prompt the user explicitly when your program executes. Most programs provide this kind of feature. E.g. in Windows from the Start..Run box, type "notepad mydoc.txt". This will start notepad and attempt to open a file named mydoc.txt without requiring you to use the GUI interface. Your OS provides this ability by parsing the command line when you start your program and passing the additional command line words as arguments: `int argc` and `char *argv[]` to the main() routine.

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

The `argc` value is an integer indicating how many command line arguments were entered (note that the executable program name is included in the count, so argc will always be at least one.) The `argv` argument is an array of character strings. For example, if we run:

```
$ ./filter input.bmp blur 3 1.5 output.bmp
```

Then

```
argc is 6
argv[0] is "./filter"
argv[1] is "input.bmp"
argv[2] is "blur"
argv[3] is "3"
argv[4] is "1.5"
argv[5] is "output.bmp"
```

Note: Numeric arguments such as "1.5" are passed in as character (text) strings and need to be converted to the appropriate numeric types before operated upon. This can be accomplished with functions like "`atoi`" (ASCII to Integer) or "`atof`" (ASCII to floating point) which are defined in `<cstdlib>` and whose prototypes are shown below.

```
// returns the integer value of the number represented by the character string "string"
int atoi(char *string);

// returns the double floating point value of the number represented by the character string "string"
double atof(char *string);
```

As an example, the "1.5" argument can be converted by:

```
double x;
x = atof(argv[3]);
```

Multi-file compilation: Most real-world programs are made up of more than one source code file and thus require the compiler to generate code and then link several files together to produce an executable. In this lab, we have provided `bmplib.h` and `bmplib.o`. `bmplib.o` is an "object" file (.o extension) representing the compiled (but

not linked) functions to perform .BMP image I/O. It is not a text file but binary instructions and memory initialization commands.

An object file can be created from a C++ file by using the -c extension to the compiler.

```
$ compile -g –Wall –c bmplib.cpp
```

This command will create the `bmplib.o`.

`bmplib.h` is a header file that includes prototypes and other declarations that you can include into your C code that will allow you to call the functions in `bmplib.o`. To compile your code with the BMP functions and then link them together you could run:

```
$ compile -g –Wall –o filter bmplib.o filter.cpp
```

You can list any number of .o files and C files on the command line. The C files will be compiled and then linked together with all of the .o files specified producing an executable as output.

'make' and Makefiles: As more files become part of your program, you will not want to compile EVERY file again when you simply make a change to one file. However, keeping track of which files have changed and thus require recompilation can also become difficult. Enter the 'make' utility. This program takes as input a text file usually named 'Makefile' which includes commands that identify the order dependencies of files on each other (i.e. if file 1 changes, then it may require re-compiling file 2 and file 3) and the commands to perform the compilation.

Typing '`make`' at the command line will compile all necessary files and produce the output executable: filter.

Need to recompile your code? Just type '`make`' again and it will only compile what you've changed.

To start fresh and remove any temporary files, type '`make clean`' followed by '`make`'.

Summary: You do not have to type in any '`compile` …' commands. Just use '`make`'.

More information about the 'make' utility and Makefiles can be found at:

http://www.eng.hawaii.edu/Tutor/Make/
http://frank.mtsu.edu/~csdept/FacilitiesAndResources/make.htm

10 Prelab

The exercises below will help you formalize some of the concepts discussed above before you start programming.

1. **Padding design:** If we restrict the size of the Gaussian kernels to odd integers between 3 and 11 inclusive, and we only allow 256x256 pixel images, what is the size of the largest padded image needed to handle padding with any kernel size? At what index will the upper-left pixel of the original image be placed in the padded image (answer in terms of N, the kernel size)? At what index in the padded array will the lower-right pixel of the original image be placed?

2. **Kernel Design:** Manually compute the raw and normalized Gaussian kernels for N=3, sigma=2. Use 4 decimal places. Discuss what would happen to the image if we used the raw kernel values.

11 Implementation

In this section we will describe the inputs and functions your program must implement at a minimum to get full credit on this programming assignment. For this assignment all BMP images will be exactly 256x256 pixels. In addition, the largest kernel used in this assignment is 11x11. Since C/C++ does not like array declarations where the size is a variable, you can declare all of your kernels with fixed 11x11 dimensions. At runtime, you will get N from the command line, so you will just fill in the upper-left NxN elements for your kernels (to create the Sobel, Gaussian, and unsharp filter values). Then, you will always pass an 11x11 kernel to your functions but those functions will only use the upper-left NxN elements.

1. Our overall program will be called `filter` and implement the three image processing filters described above, allowing the user to choose any 1 filter per run of the program. Your program will take command line arguments to control which filter is applied to the input image:
   a. `./filter <input file name> dummy <output file name>`
   b. `./filter <input file name> sobel <output file name>`
   c. `./filter <input file name> blur <kernel size N> <sigma> <output file name>`
   d. `./filter <input file name> unsharp <kernel size N> <sigma> <alpha> <output file name>`

   `<input file name>` will be a string like '`tommy.bmp`'
   The 2nd argument will be the filter to apply: `dummy`, `sobel`, `blur` or `unsharp`
   `<kernel size N>` is the size of the kernel to create for the Gaussian (min=3, max=11)
   `<sigma>` is the variance parameter in the Gaussian (must be non-zero)
   `<alpha>` is the mix-in parameter in the unsharp equation (0 < alpha <= 1.0)

2. We have completed the **main** function for you. But you should read and understand what it is doing. In particular, **main** performs the following tasks:
    a. Checks that a minimum amount of arguments are provided.
    b. Attempts to open the input file and read in the input image data to an array: `unsigned char input[SIZE][SIZE][3]`. If the file is unable to be opened, the program will quit.
    c. Checks the 2nd parameter is one of `dummy`, `sobel`, `blur` or `unsharp`. **Based on this argument we determine what other command lines arguments you should expect and check.**
    d. We then convert the appropriate command line arguments to the appropriate data type and invoke functions to carry out the specific task.
3. As you create the output image, place it in an array `unsigned char output[SIZE][SIZE][3]`. `main` will write that array to the output file name.
4. The **`dummy`** function is complete. <u>You do not need to modify it</u>. It shows you an example of how to declare a kernel and prepare it for use with `convolve`. It then calls `convolve` to generate the output image.
5. **<u>Complete</u>** the function `convolve(unsigned char out[][SIZE][3], unsigned char in[][SIZE][3] , int N, double kernel[][11])` that takes an output array, an input array and a generic kernel of which only NxN is used out of the 11x11 total size. This function should perform the convolution operation of the kernel with the image. Inside this function is where the image will need to be padded. Clamping can also be performed.
6. **<u>Complete</u>** the function: `void sobel(unsigned char out[][SIZE], unsigned char in[][SIZE])`. It is started for you.
    a. You should convolve each of the horizontal direction kernels one at a time producing two resulting arrays.
    b. Then add the two results together (pixel-by-pixel) to produce the final output image.
7. **<u>Implement from scratch</u>** a function `gaussian(k[][11], int N, double sigma)` that fills the upper NxN elements of the 11x11 kernel with the normalized Gaussian. This function only generates the kernel to be used. It does NOT actually perform the filtering/convolution. This function should print the kernel to the screen in a nice 2D table format on the screen. This will help you ensure you've computed things correctly.
8. **<u>Implement from scratch</u>** the **`gaussian_filter`** function. To do so use the `gaussian()` [to produce the kernel] and `convolve()` functions from the given command line parameters. **`gaussian_filter`** should be a function with the prototype `gaussian_filter(output [][SIZE][3], input[][SIZE][3], int N, double sigma)`. N and sigma have the same meaning as for the `gaussian()` function.
9. **<u>Implement from scratch</u>** the **`unsharp`** mask filter using the `gaussian()` and convolve() functions, along with the command line parameters. This should be a function with the prototype `unsharp(output[][SIZE][3], input[][SIZE][3], int N, double sigma)`.
10. Compile your program fixing any compile time errors.
11. Run your program on various inputs. Here are some sample command lines:

a. `./filter tommy.bmp blur 5 2.0 tommy_blur.bmp`
b. `./filter bw_tile_wikimedia.bmp sobel bw_tile_sobel.bmp`
c. `./filter usc_ucla_wikimedia.bmp unsharp 5 2.0 0.7 usc_ucla_unsharp.bmp`

12 Implementation Hints

These are a few hints to help you get started:

- Do not try to write the whole program start-to-finish. Write the program in stages. For example, you can initially write your convolve() function so it doesn't actually do anything besides copy input to output. Then you can get your program working so it opens a BMP, calls convolve() and then writes the output file. This will get you practice passing BMPs as arrays as well as handling the command line arguments. Once this is working, move on to actually implementing the other functions.
- Pixel math: pixels in our images are represented by unsigned char with values from 0-255. When implementing the filters you may need to cast the pixel values to doubles before performing the mathematical operations. Be sure to check the resulting pixel values lie in the range 0-255 before casting back to unsigned char. If a pixel value is negative, set it to 0 and if it is above 255 set it to 255. This step is known as clamping.
- Make sure your code is commented well. This will help not only you, but the graders.

13 Experimentation

Do the following experiments and comment on the results in your readme.txt

- Filter the same image with the Gaussian blur filter while varying N and sigma. If you hold N constant and vary sigma, what do you see? Conversely, if you vary N and hold sigma the same, what do you see?
- Filter a few images with the Sobel filter? What does the Sobel filter appear to 'do'?
- The Gaussian blur filter and the unsharp mask filter can be thought of as inverses of each other. Blur an image with the Gaussian blur and then attempt to un-blur it using unsharp-mask. Do you get the original image back? Provide a 2-3 sentence explanation for why you do not recover the original (i.e. they are not inverse operations).

14 Troubleshooting

Before you post a question or seek help make sure you:

1. Can view the images you downloaded. If you have problems with eog make sure you are in the right directory where your images are located.

2. Try printing out some pixel values to ensure you are reading in the input images correctly. Just be warned if you want to print out the numeric values of pixels, you'll need to cast to an integer first, as in:

```
cout << (int)inputImage[0][0][0] << endl;
```

   This is because otherwise, the image arrays are unsigned chars which will print out as actual ASCII characters.

3. Try using a debugger to run through and examine your values. When you want to debug a program that needs command line arguments (like ours which requires you to pass it the name of the input files, the threshold, etc.) when you start the program, as in:

```
$ ./filter tommy.bmp sobel output.bmp
```

   You start gdb normally with just the program name:

```
$ gdb filter
```

Set any breakpoints you desire.
Then when gdb starts up, after `run`, you also type in the command line argument:

```
run tommy.bmp sobel output.bmp
```

15 Readme.txt

In addition to your name and e-mail your "readme.txt" file shall include the answers to the **prelab questions** and the questions found under “Experimentation.” Also answer the following:

- Express in mathematical terms how the number of calculations your program does grows with the size, N, of the kernel.

16 Submission

Submit your 'filter.cpp' and 'readme.txt' files to the class website.