CSE 190: Assignment 1—Image and Signal Processing

Ravi Ramamoorthi

The purpose of this assignment is to implement some basic image processing operations, such as those found in programs like *Photoshop* or *ImageMagick*. This includes basic things like brightening, gamma correcting and cropping, as well as more advanced operations like antialiased shifting and resizing. The best reference will be the material in lecture. The assignment description is intended to be fairly complete in terms of specifying what you need to do.

1 Turning in the Assignment

You will turn in the assignment by creating a website. This should include a complete description or writeup showcasing your results. For each implemented operation, show us 2 to 3 examples of applying that operation with different parameters on at least one image. For completeness, you should indicate any nonstandard implementation issues you found, any hardcoded parameters etc. This is part of the assignment; some points are allocated for the quality of the writeup. Failure to fully document the results of your program will make the graders very unhappy. Being honest, noting what works and what doesn't work for all the features will make the assignment easy to grade and make us very happy. That said, the purpose of the assignment is to have fun; we will try not to be too harsh on technicalities. Your website should also include downloadable executable code. We may request you for source code if required to evaluate your program but *please do not* post your source code publicly on your website or on public github repositories. *It is also not permitted to copy source code from previous instances of this or a similar class.* Do not modify the website after the due date of the assignment. Also, send an e-mail to the instructor/TA before the due date with the link to your website, and a zip file containing your source and executables. Please note that many mailing programs do not allow *.zip* files; if this is an issue, please rename to *.foo*

You may (and are encouraged to) work in groups of two for the assignment. Only one assignment needs to be handed in. The requirements remain the same if you work alone, though we will give some allowance in terms of assigning the final grades if you need to work alone.

2 Support

For your benefit, we provide skeleton code for parsing arguments, so you need only worry about the actual image processing. It is *not* required to use the skeleton code we provide, and you could write the entire implementation from scratch if you prefer, but you must then implement equivalent support functionality.

In principle, you should only need to add code to the regions indicated as such, though you will need to understand the skeleton at some level, and it is ok to modify it if you feel that to be necessary. To keep things simple, everything will be executed on the command line; there is no user interface or OpenGL required for this assignment. By command line, we mean the old *cmd.exe* (DOS shell) in Windows (or equivalent under Visual Studio), not the new Power Shell. For MacOS users, we mean a standard Unix Terminal. We work only with Windows uncompressed 24 bit bmp files. Some are provided with the assignment. You can obtain others by googling or converting from other formats. The important files included in the skeleton are:

1. *main.cpp* parses the command line, calling the functions you write
2. *image.cpp , image.h* Image processing with stubs for the assignment
3. *pixel.cpp , pixel.h* Pixel processing functions
4. *vector.cpp, vector.h* A simple 2D vector class that you may or may not find useful
5. *bmp.cpp, bmp.h* Functions to read and write Windows bmp files that you probably don't need to bother about (which is the main point of providing you this skeleton code)
6. *Makefile* A Unix Makefile, that should also work from the command line for MacOS (Unix/MacOS only).

Please note that the images, source and headers are all in their individual sub-directories, and for Windows, a Visual Studio Project is provided. If you run into difficulties, please contact the instructor or TA/tutor immediately.

The first step after downloading the files is to make sure the assignment compiles and runs on your platform. You might try writing the simplest filter (probably gamma correction) to make sure you won't run into trivial technical problems later on.

In addition, we provide our example solution (this is an old Windows exe; it should be able to be run in emulators like wine on other platforms). Please note the correct way to run this in Windows (courtesy of a student in last year's class): (1) Run the *cmd* console in windows (old DOS shell). (2) Using the command *cd*, switch to the folder where *solution.exe* is located at. (Another trick instead of step 1 and 2 is to press shift then right click on the folder, select "Open Command Window here"). (3) Type *solution.exe -gamma 1.7 < wave.bmp > out.bmp* or whatever operation you want to run. Make sure the image files are in the same folder of solution.exe.

An *important caveat* is that image processing programs of this complexity are unlikely to agree completely, given the number of assumptions and implementation issues involved. So, take the solution as a guideline; we will not be harsh on technicalities. Also, it is entirely possible that the sample solution has bugs. (In particular, there is a minor bug with random dithering that was pointed out last year). Any group pointing out a new significant bug (and providing a correct implementation) will be awarded 2 points for each such bug report.

The user interface for the program is simple and works as follows:

1. *image -gamma 1.7 < in.bmp > out.bmp*
   This calls the program with the argument *-gamma* for gamma correction with parameter 1.7. The input image (in.bmp) is taken from standard input and the output is written to standard output (hence redirected here to out.bmp).
2. *image -help*
   Lists all of the commands and the arguments used for them. This is also documented in the ensuing sections that define the assignment.
3. *image -gamma 1.5 -size 300 300 < in.bmp > out.bmp*
   In general, commands are processed in the order they are found. For example, this first gamma corrects the image and then resizes it.

Acknowledgement: The skeleton code for this assignment (and some parts of the requirements) is adapted from a similar assignment developed for Princeton's COS 426 graphics course.

3 What you need to implement

You need to implement a number of image processing functions that have been left blank in the skeleton code. This section defines the assignment carefully, along with point values allotted to each section. In general, get the simplest filters working first (the assignment is loosely ordered by the difficulty of the filters) and gradually get the more complicated ones working. Remember to START EARLY. You have plenty of time to do the assignment, but it might become difficult if you start at the last moment, since there are a fair number of requirements. You are also encouraged to carefully read through the whole assignment first before starting, especially if you do the requirements out of order.

3.1 General: Writeup and Program Behavior (7 points)

7 points in this assignment will be allocated for a simple and clear writeup. You will also lose points if your program behaves poorly. By this I mean you should gracefully terminate rather than crashing when presented input you are not expecting. It is ok to make assumptions about the input or correct it if wrong in boundary cases (blur width larger than image size, even size kernels rather than odd, unreasonable parameter values), but you should try to do something reasonable or terminate gracefully with an error message rather than crash. We are not going to try deliberately to crash your program or look to be harsh on the writeup, but the above are simply good design principles, to also be followed for the remainder of the course.

3.2 Basic Operations (10 points)

You should implement the following very basic image processing operations. You are encouraged to get these simple examples working first before starting on the more complicated examples described next. Getting these working will also give you an idea of some of the skeleton code and how some of the built in pixel processing functions can be useful. In general, for all of the operations in this assignment, except edge detection, and to some extent, finding luminance values in the examples below, you will operate independently on the three color channels, red, green and blue.

1. **Image::Brighten (double factor)**
   image -brightness $< factor >$

   This brightens the image. The factor should be non-negative. A factor of 1.0 preserves the current image, 0.0 makes it black and other factors simply scale the pixel values up (darkening for values less than 1 and brightening for larger values). One way of looking at this is as linear interpolation with a black image. Consult the Graphica Obscura website at *http://www.graficaobscura.com/interp/index.html* for a description of this and the next two operations as interpolation and extrapolation. Your writeup should include a sequence of images (along with the original) for different parameters applied to the same image, as on the graphica obscura website.

2. **Image::ChangeContrast (double factor)**
   image -contrast $< factor >$

   This changes the contrast of the image. Again, refer to the Graphica Obscura website. You must first compute a constant gray image with the average image luminance (note the pixel Luminance function provided in the skeleton code; this is also generally useful for converting a color image to grayscale). Then, you simply need to interpolate from this gray image instead of black as in the previous example. A parameter value of 0 produces a grey image with no contrast, 1 gives the original image, between 0 and 1 loses contrast, greater than 1 increases contrast, and less than 0 (unlike the previous example, this part allows for negative values) inverts the image. Once again, look at the graphica obscura website for an example of what your writeup should look like.

3. **Image::ChangeSaturation (double factor)**
   image -saturation $< factor >$

   This is very similar to the previous example, except that the grey image used for interpolation is no longer constant, but is simply a grayscale version of the original. For values between 0 and 1, we make the image grayer, reducing the saturation of the colors. For larger values, we extrapolate increasing

saturation. And for negative values, we invert the hues or colors of the input image, like a photographic negative. Once again, refer to the graphica obscura website, producing images and a writeup like it.

4. **Image::ChangeGamma (double factor)**
   image -gamma $< \gamma >$

   This implements gamma correction of the image. For each pixel, you simply need to implement the formula $C_{new} = C_{old}^{1/\gamma}$, where C_{new} is the new color and C_{old} is the old or original color. The value of the gamma factor should be positive. A value of 1 leaves the image unchanged, values greater than 1 brighten it, and lower values darken it. Gamma correction of images is needed to account for nonlinearities in the display device and was discussed in class.

5. **Image::Crop (int x, int y, int w, int h)**
   image -crop x y w h

   This allows one to crop the image, for instance to focus on someone's face. We assume that x ranges from 0 to width-1 from left to right and y from 0 to height-1 from top to bottom. All parameters must be positive. w and h are the size of the cropped image. Show at least one example of cropping an image to focus on some interesting aspect in your writeup.

3.3 Quantization and Dithering (10 points)

In this part of the assignment, you will explore various strategies for quantizing images with a small number of bits. These techniques are used for instance in printing or in limited color displays. Ideally, your writeup should include a 3×5 table for an appropriate image. The rows of the table represent the 3 quantization and dithering methods below. The columns of the table represent the number of bits used (going from 1 to 5 is reasonable). You should clearly see contouring for lower numbers of bits. Random dithering and especially Floyd-Steinberg should perform significantly better than simple quantization.

1. **Image::Quantize (int nbits)**
   image -quantize $<$ *nbits* $>$

   This quantizes the image to use *nbits* per color channel (where *nbits* lies between 1 and 8). Your writeup should include examples of an image quantized with increasing numbers of bits. As you decrease the number of bits, you will see visible contouring as one transitions from one quantum to the next.

   To quantize, you need to define a map between the input [0-255] and the output [0-$2^{nbits}-1$]. A simple way of doing this is to first convert all values into floating point to lie between 0 and 1 (by dividing the input by 256). Then, select the quantum using

   $$q = \text{floor}(p * b),$$

   where q is the appropriate quantum, p is the original pixel value in the range from 0 to 1, and b is the number of bins or quanta $b = 2^{nbits}$. These values must then be mapped back into the $0 - 255$ range, which can be done by computing the final color

   $$c_f = \text{floor}(255 * q/(b-1)).$$

   As an example of this, consider quantization to 3 bits. In this case, $b = 8$. Given an initial input color c, we first compute the required quantum as $q = \text{floor}(c * 8/256) = \text{floor}(c/32)$. Thus, the range of the image is divided into steps of size 32, with each element being mapped to one of the quanta from 0 to 7. The final color is then given by $c_f = (255/7) * \text{floor}(c/32)$. The factor of $255/7$ converts the quanta from 0 to 7 to the range from 0 to 255 for display. Note that there are only 8 discrete intensity values in the output, and all 255 original input colors are mapped to one of these 8 values.

   This is only one possible mapping. It is possible to have fancier mappings that quantize non-linearly, accounting for gamma issues and nonlinear response in the eye. If you do something nonstandard, please document it in your writeup.

2. **Image::RandomDither (int nbits)**
   image -randomDither < nbits >

   Dithering adds some noise before quantizing. Thus, while we must still quantize, the contouring is somewhat broken up by noise. Perceptually, noise is found to be preferable in most cases. In the above, when you compute the quantum q, add some noise in the range from $[-.5... + .5]$. That is, $q = \text{floor}(p * b + \text{noise})$. The noise is added using the random function.

   Please note that there is a trick to this part, in that it is not properly implemented in the solution program, and so your result will not match it perfectly. (If you are adventurous, you may want to try to determine what the solution is doing wrong; Hint: Is noise added before or after quantization).

3. **Image::FloydSteinbergDither (int nbits)**
   image -FloydSteinbergDither < nbits >

   This is the most advanced algorithm, and is an error diffusion method. The error made in quantizing a particular pixel is diffused out to pixels later in the process. Specifically, the error is split and diffused out with the weights

$$\begin{aligned} \text{RIGHT}: &\quad \alpha = 7/16 \\ \text{BOTTOMLEFT}: &\quad \beta = 3/16 \\ \text{BOTTOM}: &\quad \gamma = 5/16 \\ \text{BOTTOMRIGHT}: &\quad \delta = 1/16. \end{aligned}$$

   Important implementation details: In this case, you should try to do something reasonable (as with much of the rest of the assignment) with edge conditions of pixels on the boundary. One reasonable approach is to treat the image as toroidal, wrapping around at the edges [this may not be practical for some error diffusion algorithms]. Another reasonable approach is to use only valid pixels, but then renormalize at the edges to prevent darkening of edge pixels. In your writeup, please explain in a sentence or two what you did for both this part, and the later parts of the assignment.

   Another important caveat is that the skeleton code deals with RGB values as unsigned chars. Since you may need negative values here, you should implement things carefully if you want to read the image in place. Another option is to include a dummy real valued array for the errors.

3.4 Basic Convolution and Edge Detection (10 points)

Here, you will implement some basic convolution filters for simple tasks. See the note above about edge conditions. You may want to add an auxiliary *Convolve* function to the image class to implement the convolutions for arbitrary filters. Alternatively, you may want to make use of the *Sample* function for this purpose. I would recommend you start on this after finishing the previous requirements.

1. **Image::Blur (int n)**
   image -blur < n >

   Blur the image by using a filter of width n. n should be an odd integer. Your writeup should show examples of an image blurred with kernels of width $3, 5, 7, 11, 13, 19$ or a similar range of values. The implementation of this is fairly straightforward. You simply need to implement discrete convolution with a blur filter. In general, for a filter f, the discrete convolution will compute

$$I_{new}(a, b) = \sum_{x,y} f(x - a, y - b) I_{old}(x, y).$$

   Of course, the summation needs only be computed for pixels within the width of the filter, centered at (a, b). This will clearly be more efficient for smaller width kernels. For larger kernels, it is more efficient to Fourier transform the image and compute the convolution as a multiplication in that domain. This is

not required in this assignment; extra credit will be available for those groups implementing convolution in the Fourier domain—if so, document it extensively in your writeup.

The only remaining question is what blurring kernel function f to use. The sample solution implements a gaussian blur. Gaussian filters are quite popular. In 1D, they are given by

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[\frac{-x^2}{2\sigma^2}\right].$$

For 2D filters, in general, one simply multiplies the components, $f(x, y) = f(x)f(y)$. The remaining item is how to choose the filter width σ. I used $\sigma = \text{floor}(n/2)/2$. The greater the filter width, the more blurring occurs.

Normalization: The normalization above is correct, but may need to be modified to account for the fact that you are using a finite approximation to the infinite gaussian. In general, normalization is very important; otherwise your image will darken or brighten as you apply the filters. If you run into this problem, check your normalization. Another trick is to simply add up the values in the filter for each pixel, and normalize appropriately (this automatically takes edge issues into account but might be inefficient). A quick check is to create a uniform grey image. Any of the filters acting on it should leave the image unchanged.

Integer arithmetic: You should implement your convolution in integer arithmetic as that would be simpler and more efficient than operating in floating point. Take your floating point filter above, and convert it to integers by dividing by the lowest element (furthest from the center) and rounding. Then, compute an overall normalization by adding up all the filter elements. At the end, you will divide by this overall normalization. For instance, for $n = 3$, the filter I get is

$$f(x, y) = \frac{1}{86}\begin{pmatrix} 1 & 7 & 1 \\ 7 & 54 & 7 \\ 1 & 7 & 1 \end{pmatrix}.$$

If you do anything nonstandard, like using a different filter, please document it in your writeup. The notes about normalization and integer arithmetic are important for all of the convolution operations.

2. **Image::Sharpen ()**
   image -sharpen

   The opposite of blurring is sharpening. The idea is to bring out high frequencies more (though we cannot hallucinate information that is missing). For this purpose, you simply need to convolve with a sharpening filter. The filter I used (this is a fixed 3×3 filter with no parameters) is

$$f(x, y) = \frac{1}{7}\begin{pmatrix} -1 & -2 & -1 \\ -2 & 19 & -2 \\ -1 & -2 & -1 \end{pmatrix}.$$

   For the writeup, simply show the results of sharpening at least one interesting image.

3. **Image::EdgeDetect (int threshold)**
   image -edgeDetect < threshold >

   The idea here is to detect edge pixels in the image, shading those and making the rest of the image black. Your writeup should indicate all parameters used and should show at least the examples of running your edge detector on the checkerboard and the wave image that are provided.

   Edge detection has been an entire research area, with many methods developed. The approach we will take is using a Sobel edge detector. First, we convolve the image with a gradient filter to detect regions of large horizontal or vertical gradients which might be edges. The Sobel filters are

$$f_{horiz}(x,y) = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \qquad f_{vert}(x,y) = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}.$$

It should be clear that these are simply finite differencing operators to find horizontal and vertical gradients. **An important caveat is that applying the results of these filters can lead to negative numbers.** It may require some implementation to be able to deal with this (the skeleton code only has unsigned chars).

After convolving, we have horizontal and vertical gradient images that we call G_x and G_y. We then find a composite gradient image magnitude

$$G = \sqrt{|G_x|^2 + |G_y|^2}.$$

In computing this magnitude, you should compute the net luminance at each pixel, from the three color channels (use the Luminance function in the pixel class).

Finally, the image is thresholded based on the parameter given, to convert it to an edge image. For purposes of debugging, you may want to visualize the gradient images separately before thresholding.

3.5 Antialiased Scale and Shift (10 points)

We will now implement two key operations—scaling the image, and shifting it by a real (non-integer) value. A naive approach, such as nearest neighbor sampling, will lead to significant aliasing and jaggies. Your writeup should include a variety of non-integer scales on the checkerboard, for instance 300×512, 512×300, 300×300 and 800×800. Remember that you need to show these results for all 3 filter types (see below). You should also show non-integer shifts on the checkerboard. For extra credit, you can create a movie showing sub-pixel shifts of the image, to see that it varies smoothly. A user interface allowing this in real time would be even cooler.

Filters: You will implement 3 filters or techniques, based on the value of *sampling_method* for both scale and resize. This variable can be set by using *image -sampling*. The first is **nearest neighbor or point reconstruction.** You simply pick the nearest point or pixel in the original image to reconstruct each pixel in the target image. The second is the **hat** filter, which varies linearly so

$$f(x) = 1 - \mid x \mid,$$

being nonzero only when $\mid x \mid \leq 1$. As usual, 2D filters are constructed by multiplying 1D filters. The third is the Mitchell reconstruction cubic filter which is slightly more sophisticated,

$$\begin{aligned} f(0 \leq \mid x \mid < 1) &= \frac{1}{6}\left(7 \mid x \mid^3 - 12 \mid x \mid^2 + 16/3\right) \\ f(1 \leq \mid x \mid < 2) &= \frac{1}{6}\left(-7/3 \mid x \mid^3 + 12 \mid x \mid^2 - 20 \mid x \mid + 32/3\right). \end{aligned}$$

Please note that this part of the assignment differs from earlier use of filters, in that we now use the *continuous* filter values; there are no longer discrete 3×3 or other kernels.

1. **Image* Image::Scale(int sizex, int sizey)**
   image -size < sizex sizey >

   The first operation is to resize the image. The resulting size is in the arguments. It may be simplest to implement four functions, for *MagnifyX, MagnifyY, MinifyX, MinifyY*. For a given size, you can call the appropriate chain of two functions (depending on magnification or minification).

Minification: In minification by a factor $s < 1$, we are replacing an image of width w by one of width sw, but conceptually with *fat* pixels of size $1/s$. Thus, each pixel must integrate over the corresponding region in the original image, using the appropriate reconstruction filter f. In other words, we compute

$$I_{new}(a) = \sum_x f(sx - a) I_{old}(x).$$

Of course, we need only sum where $|sx - a|$ is smaller than the filter width. That is the filter can be considered to be centered at a/s and has width $1/s$ times the normal filter width. In this case, since the filter is not necessarily evaluated at an integer location, we will need to use a parametric form, so you will **need to use floating point arithmetic rather than integer**.

Magnification: Magnification with $s > 1$ simply requires interpolating between samples in the original image to obtain a higher resolution image. We can compute it in much the same way as above, using

$$I_{new}(a) = \sum_x f(x - a/s) I_{old}(x),$$

where the filter width is now the original width. The same filters as for minification should be used.

2. **Image::Shift(double sx, double sy)**
   image -shift < sx sy >

   This function shifts the image, based on values of sx and sy. You should check for integers, and implement different integer and real-valued shifts. For the real-valued shift, one simply needs to use the formula

$$I_{new}(a, b) = \sum_{x,y} f(x - a + sx, y - b + sy) I_{old}(x, y).$$

   As before, you need to implement all three filters.

3.6 Fun nonlinear filters (3 points)

Fill in the *fun* filter to be some kind of nonlinear or non-photorealistic mapping. The goal should be to produce an interesting image. Extra credit of up to two points will be awarded to the most aesthetic or artistic images so produced. Your writeup should document briefly what you did. This part of the assignment is open-ended and so not spelled out in much detail. You can consider projections like fisheye, simple image rotation or more complicated things. What you do is up to your imagination and initiative.

3.7 Extra credit: Compositing and Morphing (10 points)

The skeleton code includes functions for image compositing and morphing. Filling these in and developing a good complete system will get you 10 points. Partial credit will be available for partial solutions. For the compositing, I expect a complete system per Porter and Duff's SIGGRAPH 84 paper. You should implement (and document in your writeup) a variety of compositing and alpha matting operations. You should also include some fun examples like pasting your photograph into an image of famous people, or something similarly interesting. For the morphing, you should implement Beier-Neely's SIGGRAPH 92 paper. You should create a video of several interesting morphs (between people works well as in the famous Michael Jackson Black or White video). This probably also requires writing at least an auxiliary program or user interface to draw the line segments. This could make a completely different program as well. To get full points for the morphing, I would like to see an interactive demo as well as a writeup documenting your user interface with screenshots.

Remember that you should approach this assignment by starting early and working through it in an incremental fashion, doing the earliest filters first. Do not attempt the extra credit until you're absolutely sure the rest of the assignment is solid. But for those of you who want to have fun, go for it.