CSCI 1133, Lab Exercise 15

Image processing is an extremely important field of applied computer science that is used in everything from self-driving cars to Snapchat. Image processing is the study of using algorithms to transform images from one form into another. This lab is designed to establish a basis to working with images, and to introduce the topic of image processing.

Note: This lab assumes you have downloaded the all the files provided on the Moodle page and put them in your working directory. It is recommended that you test your algorithms with `tiny.png` and `tiny2.png` before using larger images! Some test images obtained from this website.

1 Warm Up

1.1 Introduction to `numpy`

This semester, you've already had some experience with nifty libraries in Python, like `pygame`. Before we get are able to work with images at all, there's an important library we need to talk about: `numpy`. `numpy` is a powerful Python library for fast operations on multi-dimensional data, like images. The central object of `numpy` is the **array**. Arrays are very similar to regular Python lists – in fact, usually a `numpy` array is created from a Python list. Enough blather, let's do some programming! As always, if you get stuck on anything ask your TAs, and if you're curious about `numpy` functions, read the documentation at `numpy.org`. Follow the steps below in the Python interactive shell to get started using `numpy`!

1. Import the `numpy` library. Python programmers usually use the following statement to abbreviate `numpy` as `np`:
   ```
   >>> import numpy as np
   ```

2. Create a new `numpy` array
   ```
   >>> numbers = np.array([1, 2, 3, 4])
   ```

3. Arrays can be iterated over in the same was as regular Python lists. Print out each item of `numbers` on a new line:
   ```
   >>> for n in numbers:
   ...     print(n)
   ```

Arrays can be nested indefinitely (creating n-dimensional arrays), but for this lab we'll be focusing on 2-dimensional structures, because that's how images are represented. If you need a refresher on how multidimensional sequential structures work, take a look at Lab 6. **Important:** The syntax for multidimensional indexing in arrays is slightly different with arrays than regular Python lists!

4. Declare a 2-dimensional matrix:
   ```
   >>> matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
   ```

5. Print out the element at index `[2, 2]`. (this is the syntax difference – note the *comma-separated* indices representing the `[row, column]` index!)
   ```
   >>> print(matrix[2, 2])
   ```

6. `numpy` arrays are *mutable*, just like regular Python lists. This means that sometimes we need to make copies of the array to avoid messing up our program! Make a copy of the 2D array

`matrix`:

```
>>> matrix_copy = np.array(matrix)
```

1.2 Images as matrices

Grayscale images are usually represented as 2-dimensional arrays of integers ranging from 0 to 255. The integer represents the *intensity value* at each pixel – 0 is black (low intensity), and 255 is white (high intensity). Color images (RGB) are represented similarly, only extended into the third dimension for the red, green, and blue channels. You are welcome to try this lab using RGB images for an additional challenge, but it is *highly recommended* that you start with grayscale – and we'll assume for the rest of the lab that all images you're using are grayscale.

`numpy` is a part of a larger suite of libraries called `scipy`. `scipy` provides several functions that will be helpful during the course of this lab.

- `imread`: Load an image as a `numpy` array
- `imshow`: Show an image array in a separate window
- `imsave`: Write an image array to disk as an image file

Let's import those functions now:

```
>>> from scipy.ndimage import imread
>>> from scipy.misc import imshow, imsave
```

Follow along in your interactive Python shell:

1. Load a small (16x16 pixel) grayscale image into a `numpy` array using `imread`.
   ```
   >>> tiny = imread('tiny.png')
   ```
2. Let's see what the contents of the array `tiny` is! You should see a 2D array of integers with values somewhere between 0 and 255!
   ```
   >>> print(tiny)
   ```
3. Print out the value of the pixel at index `(3, 14)`. (don't forget the comma-based indexing!)
   ```
   >>> print(tiny[3, 14])
   ```
4. Load another test image in, and show it on the screen using `imshow`. You should see the picture pop up in a separate window!
   ```
   >>> pixels = imread('cat_8bit.png')
   >>> imshow(pixels)
   ```
5. Now, write the image array `pixels` as a new image file. If you open a file manager in your working directory, you should see it appear and then be able to open it in a photo preview application!
   ```
   >>> imsave('cat_copy.png', pixels)
   ```
6. (Optional) If you want to download any images from the Internet to use during this lab and convert them to grayscale, you can provide an additional argument `flatten=True` to `imread`. Try it with `cat_rgb.png`, a color image:
   ```
   >>> pixels = imread('cat_rgb.png', flatten=True)
   ```

1.3 Image Clamping

As stated previously, images can only have values in the range [0, 255]. However, oftentimes we'll do a lot of math with the intensity values in an image, which results in values that are outside that range! This is a problem, and the solution is called *clamping*. Clamping takes all "invalid" values in an image and restores them back into the range [0, 255]. All values above 255 become 255, and all values below zero become zero.

★ Create a function called `clamp(image)` that will take a single image array as an argument, and return a new (copied) image where every pixel is clamped to be in the range [0, 255].

Hint: Most image processing code in this lab will be very similar to code you've written before dealing with 2-dimensional matrices. Usually, your code will look something like this:

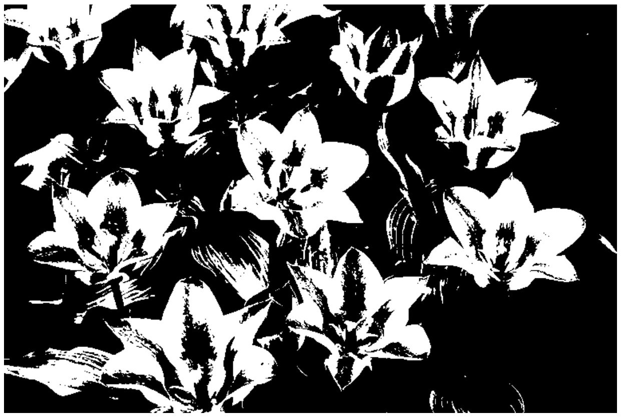
```
def do_something(im):
    rows, cols = im.shape # find size of image
    new_im = np.array(im) # copy image
    for row in range(rows):
        for col in range(cols):
            # do something with each pixel
    return new_im
```

2 Stretch: Linear filters

Now that we've gotten a good idea of how images are represented in Python, we can begin to actually do image processing! Let's begin with **Linear Filters**. A linear filter iterates through the pixels of an image, and performs a math operation on each pixel.

2.1 Simple Thresholding

Original image


Thresholded image ($t = 128$)

Simple thresholding separates an image by light and dark pixels, and returns a *binary image*. Binary images only contain two values (0 and 255). The user (or programmer) chooses a threshold value t, then applies the following function to every pixel x:

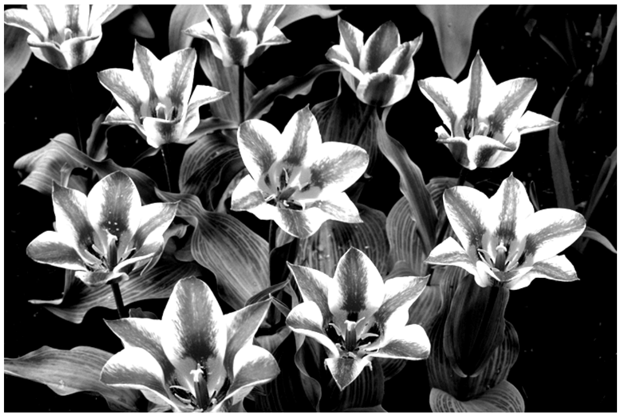
$$f(x) = \begin{cases} 255 & x > t \\ 0 & x <= t \end{cases}$$

The resulting image after filtering has only the values 0 and 255. The high intensity values (above threshold `t`) become 255, and the low intensity values (below or at `t`) become 0.

★ Create a function `threshold(im, t)` that takes an image and a cutoff value t as arguments, and returns a binary image thresholded at the given value. Keep in mind that you *should not* alter the image from the argument; create a copy, modify it, and return the copy.

2.2 Contrast Adjustment

Original image


Contrast-adjusted image ($amount = 100$)

An essential feature for any photo editing software is **contrast adjustment**. Sometimes, when you take a photo, the details aren't vibrant enough, or they're too vibrant, and contrast adjustment can help mitigate that issue. Contrast adjustment is actually just another linear filter!

There are three main steps to the contrast adjustment algorithm:

1. Calculate the contrast factor
2. Loop through all pixels and adjust using contrast factor
3. Clamp the results from the contrast adjustment

The contrast correction factor can be calculated using the formula

$$factor = \frac{259 * (amount + 255)}{255 * (259 - amount)}$$

where $amount$ is the desired contrast adjustment, in the range [-255, 255]. The contrast factor can be used to calculate the new pixel value $f(x)$ from the original pixel value x:

$$f(x) = factor * (x - 128) + 128$$

★ Create a function `contrast(im, amount)` that adjusts the contrast of an image pixel by pixel, using the formulae above. **Hint:** You may find it useful to use the `clamp` function that you created in the warmup to obtain the final pixel values.

Inspiration for these formulae was drawn from this website.

Original image

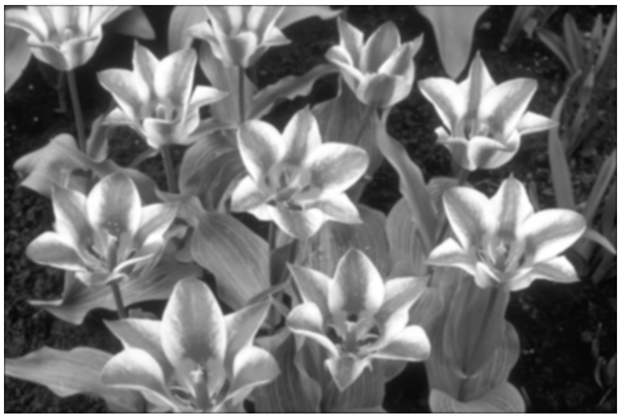


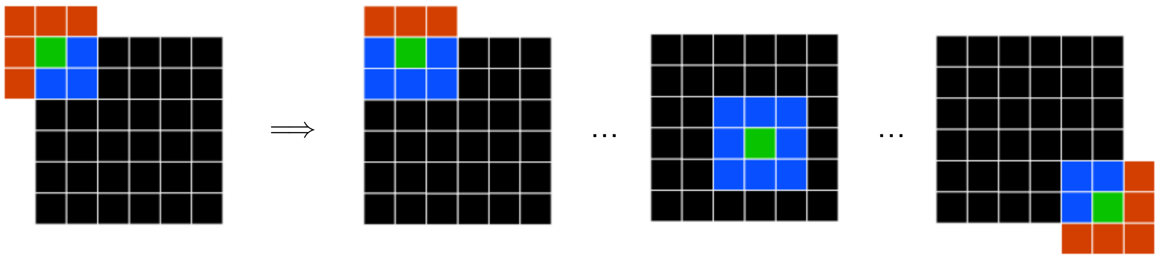
Image blurred three times with 3x3 mean filter

3 Workout: Convolution Filters

3.1 Mean (Blur) Filter

Another useful filter that all respectable photo editing software has is a good “blur” (or “mean”) filter. Mean filters make all hard edges softer in an image, and can also help to remove salt-and-pepper noise (speckling) on an image. Mean filters are a type of **convolution filter**. Like linear filters, convolution filters iterate through every pixel, but convolution filters also consider the *neighbors* of each pixel.

Conceptually, convolution filters can be thought of as a “moving window,” or “kernel” sliding over every pixel of an image (highlighting which pixels of the moving window are actually inside the image, and the center of the moving window):


3x3 moving window sliding over an image

The moving window for mean filters look like this:

1	1	1
1	1	1
1	1	1

This means that mean filters consider every pixel in the *8-connected* neighborhood of every pixel (row, col). The pixel (row, col) is the center element of the moving window. Mean filters take the sum of all 9 pixels in the moving window, then divide by 9 to obtain the new value for the pixel at (row, col).

Warning: Be careful not to go off the edge of the image! It’s a good idea to check if your current (row, col) is actually inside the image ($row \geq 0$ and $row < rows$ and $col \geq 0$ and $col < cols$).

★ Create a function `mean_filter(image)` that takes in an image as an argument, and returns a new image that is blurred using the mean filter technique. **Note:** If you’re using a large image, you may need to blur several times to have a noticeable difference.

3.2 Sobel Edge Detection

Original image

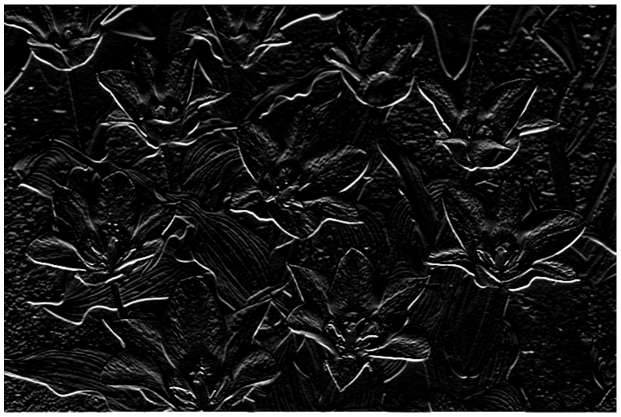


Image with Sobel y-direction edge detection

Edge detection is another useful image processing technique. As the name implies, it will extract the edges of objects that it finds in the image. Believe it or not, edge detection is actually just implemented as another convolution filter! The moving windows for Sobel edge detection in the x and y direction (respectively) are modeled by:

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

(x moving window)

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

(y moving window)

You can declare these moving windows in Python as 2D `numpy` arrays.

The code for edge detection will look similar to that from your mean filter, but there are a few additional nuances to worry about. At each pixel (row, col):

1. Initialize a running sum for this pixel to zero
2. Multiply each of the pixel's neighbors by its corresponding element in the moving window, and add the result to the running sum. (remember, the pixel (row, col) corresponds to the *center* of the moving window)
3. Set the new image's value at (row, col) to be the running sum

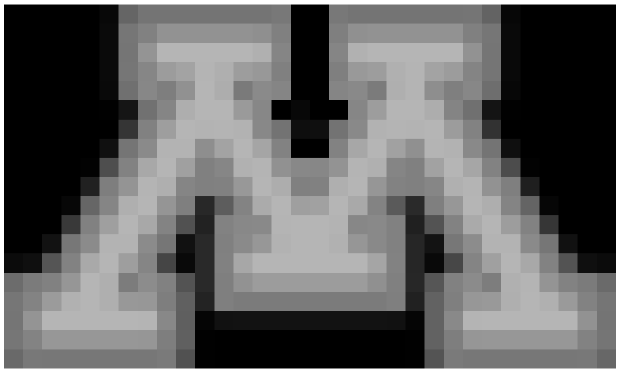
Once again, we're doing some math with pixel intensities in this algorithm, so there's a chance that we may overflow the $[0, 255]$ bounds — use the `clamp` function.

★ Create a function `sobel_edge(image)` that takes in an image as an argument, and returns a new image that has been edge-detected in either the x or y direction (you can choose which one to implement). Don't forget to make sure you're inside both the image and the moving window! (you may need multiple if statements to check that).

4 Challenge

4.1 ASCII Art

ASCII Art is a fun topic that goes back to the early days of computing when there were no computer graphics or user interfaces, and everything was just characters. ASCII Art is the painstaking,


Original image

```
    J888888888) J888888888)
    J888888888) J888888888)
    J888888888) J888888888)
    J888888888) J888888888)
    J88888888P ,'788888888)
    '78888888 L (L8888888P.
    "L88888888P-788888888 ,
   "L888888888 (L888888888 ,
   J88888888888888888888888)
  "L88888888888888888888888 ,
  J8888888888888888888888888)
 "L8888888888888888888888888 ,
(L888888888888888888888888888 (
8888888888888888888888888888888
8888888888888888888888888888888
8888888888888888888888888888888
8888888888P---------78888888888
8888888888)         J8888888888
```

ASCII Art

meticulous art form of translating real world content or images into pictures made up of only characters. Today, we'll make the computer do all the hard work and automatically generate ASCII Art from images that we feed it. The algorithm works as follows:

1. Threshold to get binary image
2. Run conversion for each pixel, matching from character template table

The ASCII image conversion algorithm we'll use is similar to the convolution filter mentioned above. It has a 2x2 moving window, which we'll pattern match with the binary image. Instead of the moving window being a matrix that we use to do math on the image, it is a "template-matching" moving window, which maps the following "templates" into characters (`.` represents 0 and `X` represents 255):

$\begin{array}{|c|c|}\hline . & . \\ \hline . & . \\ \hline\end{array} \Longrightarrow$ [space] $\begin{array}{|c|c|}\hline . & X \\ \hline . & . \\ \hline\end{array} \Longrightarrow$ ' $\begin{array}{|c|c|}\hline X & . \\ \hline . & . \\ \hline\end{array} \Longrightarrow$. $\begin{array}{|c|c|}\hline . & . \\ \hline X & . \\ \hline\end{array} \Longrightarrow$,

$\begin{array}{|c|c|}\hline . & . \\ \hline . & X \\ \hline\end{array} \Longrightarrow$ ” $\begin{array}{|c|c|}\hline X & X \\ \hline . & . \\ \hline\end{array} \Longrightarrow$ - $\begin{array}{|c|c|}\hline . & . \\ \hline X & X \\ \hline\end{array} \Longrightarrow$ ($\begin{array}{|c|c|}\hline X & . \\ \hline X & . \\ \hline\end{array} \Longrightarrow$)

$\begin{array}{|c|c|}\hline . & X \\ \hline . & X \\ \hline\end{array} \Longrightarrow$ J $\begin{array}{|c|c|}\hline . & X \\ \hline X & X \\ \hline\end{array} \Longrightarrow$ L $\begin{array}{|c|c|}\hline X & X \\ \hline . & X \\ \hline\end{array} \Longrightarrow$ 7 $\begin{array}{|c|c|}\hline X & X \\ \hline X & . \\ \hline\end{array} \Longrightarrow$ P

$\begin{array}{|c|c|}\hline X & X \\ \hline X & X \\ \hline\end{array} \Longrightarrow$ 8

You may find it helpful to use the following dictionary to help convert image pixels to characters. Keys are flattened 2x2 arrays of the moving window matrices above:

```
patterns = {
        '....':' ',
        '.X..':'\'',
        'X...':'.',
        '..X.':',',
        '...X':'"',
        'XX..':'-',
        '..XX':'(',
        'X.X.':')',
        '.X.X':'J',
        '.XXX':'L',
        'XX.X':'7',
        'XXX.':'P',
        'XXXX':'8'
}
```

★ Create a function `img_to_ascii(im)` that takes in a grayscale image, and returns a list of strings that represent the ASCII art image. This problem may be easier to think about if you make a separate helper function `match_char(im, row, col)` which finds the ASCII character of a single pixel at (row, col) in an image.

Inspiration for this section was from here

★ If you're feeling up for an extra challenge, try implementing a grayscale ASCII art translator. You may find the link above useful for this – they don't give an algorithm at all, but they explain a bit more about the concept.