Image Convolution

Dr. Tom Nurkkala

September 23, 2018

Contents

1 Introduction **1**

2 Code **1**

2.1 Headers 2
2.2 Macros 2
2.3 Usage 3
2.4 Command-Line Arguments 4
2.5 Types 5
2.6 Image I/O 6
2.7 Normalization 7
2.8 Convolution 8
2.9 Convolution Kernels 9
2.10 Main 10

1 Introduction

Image convolution applies a 2D *kernel* to each pixel of a 2D image. Consider an $n \times n$ kernel k applied to input image f to create output image g, The value at pixel $g(x, y)$ corresponding to input pixel $f(x, y)$ is given by the following expressions.

$$n_2 = \lfloor n/2 \rfloor$$

$$g(x, y) = \sum_{i=-n_2}^{n_2} \sum_{j=-n_2}^{n_2} k(n_2 + i, n_2 + j) f(x - i, y - j)$$

2 Code

This section details a *sequential* implementation of image convolution.

2.1 Headers

We include some standard library headers (`stdio`, `stdlib`, and `unistd`).

The library we use for loading and saving PNG files is Lode PNG. For both loading and saving PNG images, LodePNG converts between a standard PNG file on disk and raw pixels in memory.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "lodepng.h"
```

2.2 Macros

The internal representation of the PNG file will be a 2-D array of pixels. Each pixel comprises four bytes, one byte each for the red, green, blue, and alpha channels of the image.

```c
#define BYTES_PER_PIXEL 4
#define RED_OFFSET 0
#define GREEN_OFFSET 1
#define BLUE_OFFSET 2
#define ALPHA_OFFSET 3

#define IMG_BYTE(columns, r, c, b)              \
    ((columns * BYTES_PER_PIXEL * r) +          \
     (BYTES_PER_PIXEL * c) + b)
#define CLAMP(val, min, max) \
    (val < min ? min : val > max ? max : val)
```

We define the total number of bytes per pixel (`BYTES_PER_PIXEL`) as well as offsets *within* a pixel for each of the four channels (`RED_OFFSET`, etc.).

According to standard C language convention, the raw two-dimensional image data are stored row-wise in memory. Thus, each pixel from the first row appears in order, followed by the pixels from the second row, and so forth. For an image with a width of w_c columns and whose pixels occupy w_b bytes, we can calculate the offset o of the byte at row r and column c as:

$$o = (w_c \times b \times r) + (w_b \times c)$$

The first term gives the byte offset of the beginning of row r, and the second gives the offset of the first byte of the pixel at column c of that row. We also want to pick out particular bytes from the pixel, so we extend this expression by adding a byte offset b.

$$o = (w_c \times b \times r) + (w_b \times c) + b$$

To retrieve pixels conveniently, we define `IMG_BYTE`, a *macro* that takes w_c, r, c, and b and returns an offset into the image. Because images will have different sizes at run time, we need to pass w_c. However, `BYTES_PER_PIXEL` is a compile-time constant, so the macro can simply refer to this value without requiring that we supply it as a macro argument.

As we process pixels, the convolution algorithm may yield new-pixel values that are outside the range that can be stored in a single byte. The `CLAMP` macro provides a convenient way to "clamp" values to be within a given range. Clamp uses the *ternary* operator:

```
a = bool_val ? b : c;
```

This operator evaluates `bool_val` as a Boolean. If its value is true, it sets `a` to `b`; Otherwise, it sets `a` to `c`. To clamp a value to the range [0..255], we'd code:

```
clamped_val = CLAMP(val, 0, 255);
```

2.3 Usage

The `usage` function displays user options and is also used by `main` to produce error messages. Note that the usage function exits the program with a return code of 1. This follows the Unix convention that zero means success and non-zero means something went wrong. If a program has more than one way to fail, specific non-zero return codes can be used to indicate *how* it failed.

```
void
usage(char *prog_name, char *msge)
{
  if (msge && strlen(msge)) {
    fprintf(stderr, "\n%s\n\n", msge);
  }
  fprintf(stderr, "usage: %s [flags]\n", prog_name);
  fprintf(stderr, "  -i <input file>   set input file\n");
  fprintf(stderr, "  -o <output file>  set output file\n");
  fprintf(stderr, "  -h                print help\n");
  exit(1);
}
```

2.4 Command-Line Arguments

When the kernel executes a program, it provides a list of command-line arguments to the start function. (In reality, the kernel supplies values to the `exec` system call, but when you invoke a command from the shell, these values come from the command line.)

A C program accesses these arguments by referring to the `argc` and `argv` arguments passed to the `main` function.

```
int main(int argc, char **argv)
```

Recall that C strings are usually declared as `char *`, a pointer to a character. Main's `argv` is a *pointer* to a `char *` that will be referenced as an array of strings. The first string (`argv[0]`) is always the name of the command, and the remaining strings in `argv` are its arguments. The `main` function also needs to know how many arguments were passed from the command line. The `argc` ("argument count") provides this value. The count *includes* the program name in `argv[0]`.

Rather than parsing command-line arguments ourselves, we use the standard library `getopt`. The function takes as arguments:

- `argc`
- `argv`
- Option string

The option string contains one or more characters that `getopt` will expect to find as command-line flags. If the flag takes an argument, the character must be followed by a colon (`:`). The following code block shows how to use `getopt`. Each time we call `getopt`, it returns the next command-line flag (character). When `getopt` runs out of command-line flags, it returns -1. Note the following:

1. We iterate over the options in a `while` loop. Each time through the loop, `getopt` processes one command-line argument.

2. The body of the loop is a `switch` statement that handles each flag in turn. If the flag has a corresponding argument (i.e., appears in the option string followed by a colon), `optarg` points to the argument.

3. The `getopt` function will invoke the `default` clause of the `switch` statement if it encounters an unknown flag. In this case, we invoke the `usage` function to give the user some help.

4. As we process flags with additional arguments (e.g., `i`), we capture the value of `optarg` in another variable (e.g., `input_file_name`). Note that the kernel will already have allocated space for the values of `argv`, so we're just pointing to the corresponding string as `getopt` finds it in `argv`.

```
char *input_file_name = NULL;
char *output_file_name = NULL;

int ch;
while ((ch = getopt(argc, argv, "hi:o:")) != -1) {
  switch (ch) {
  case 'i':
    input_file_name = optarg;
    break;
  case 'o':
    output_file_name = optarg;
    break;
  case 'h':
  default:
    usage(argv[0], "");
  }
}

if (!input_file_name) {
  usage(argv[0], "No input file specified");
}
if (!output_file_name) {
  usage(argv[0], "No output file specified");
}
if (strcmp(input_file_name, output_file_name) == 0) {
  usage(argv[0], "Input and output file can't be the same");
}
```

After command-line parsing is complete, we do some validation of the input parameters that we have captured. The `usage` function takes an error message, which it will print along with the help message.

2.5 Types

Each raw pixel is an `unsigned char`. We use `typedef` to create a new type for a pixel, `pixel_t`.

Typedef `image_t` is a custom type that stores an image's raw pixels with its width and height.

Finally, a convolution kernel is a 3×3 array of integers. Although pixel values will always be in the range [0..255], the factors in a kernel can be positive or negative.

```
typedef unsigned char pixel_t;
```

```

typedef struct {
  pixel_t *pixels;
  unsigned int rows;
  unsigned int columns;
} image_t;

#define KERNEL_DIM 3
typedef int kernel_t[KERNEL_DIM][KERNEL_DIM];
```

2.6 Image I/O

The function `load_and_decode` loads a PNG file from `file_name` and stores it in the address `image`. The caller will normally declare a vartiable of type `image_t`, and pass its address to this function. Note that memory will be allocated by LodePNG to store the image pixels. To avoid leaking memory, you should alays invoke `free_image` when done with an image. Similarly, `encode_and_store` stores a raw image into a PNG file.

```
void
load_and_decode(image_t *image, const char *file_name)
{
  unsigned int error =
    lodepng_decode32_file(&image->pixels, &image->columns,
    ↪  &image->rows, file_name);
  if (error) {
    fprintf(stderr, "error %u: %s\n", error,
    ↪  lodepng_error_text(error));
  }
  printf("Loaded %s (%dx%d)\n", file_name, image->columns,
  ↪  image->rows);
}

void
encode_and_store(image_t *image, const char *file_name)
{
  unsigned int error =
    lodepng_encode32_file(file_name, image->pixels, image->columns,
    ↪  image->rows);
  if (error) {
    fprintf(stderr, "error %u: %s\n", error,
    ↪  lodepng_error_text(error));
  }
```

```
  printf("Stored %s (%dx%d)\n", file_name, image->columns,
    image->rows);
}
```

The `init_image` function initializes an `image_t` structure. It dynamically allocates storage for the raw pixels of an image of the given size. This function is primarilly intended to allocate a new image with the given dimensions. When loading from a file, `load_and_decode` takes care of setting up the `image_t` structure; there is no need to call `init_image`.

When you are done using an `image_t`, free its memory with `free_image`.

```
void
init_image(image_t *image, int rows, int columns)
{
  image->rows = rows;
  image->columns = columns;
  image->pixels = (pixel_t *)malloc(rows * columns *
    BYTES_PER_PIXEL);
}

void
free_image(image_t *image)
{
  free(image->pixels);
}
```

2.7 Normalization

For each convolution kernel, we compute a value that's used as a divisor to "normalize" the new pixel value. This normalization value is simply the sum of all the values in the kernel. If the value is zero (which would make a poor divisor), we set it to one.

```
int
normalize_kernel(kernel_t kernel)
{
  int norm = 0;

  for (int r = 0;  r < KERNEL_DIM;  r++) {
    for (int c = 0;  c < KERNEL_DIM;  c++) {
      norm += kernel[r][c];
    }
  }
```

```
  if (norm == 0) {
    norm = 1;
  }

  return norm;
}
```

2.8 Convolution

The main loops of the convolution iterate over each row, column, and byte in the input image. To avoid boundary conditions, this implementation ignores the outermost pixels on all sides of the input image. This explains the form of the two outermost `for` loops:

```
for (int r = 1;  r < rows - 1;  r++)
```

In order to access the proper elements of the kernel, we calculate `half_dim`, which is half the dimension of each side of the (square) kernel. We also get the kernel normalization value.

The three nested `for` loops iterate over the byte values of each pixel in each row and column of the image. For each pixel, we calculate the `value` of the corresponding pixel in the output, which is then assigned to the `output` image.

```
void
convolve(image_t *output, image_t *input, kernel_t kernel)
{
  int columns = input->columns;
  int rows = input->rows;
  int half_dim = KERNEL_DIM / 2;
  int kernel_norm = normalize_kernel(kernel);

  init_image(output, rows, columns);

  for (int r = 1;  r < rows - 1;  r++) {
    for (int c = 1;  c < columns - 1;  c++) {
      for (int b = 0;  b < BYTES_PER_PIXEL;  b++) {
        int value = 0;

        «calculate value»

        output->pixels[IMG_BYTE(columns, r, c, b)] = value;
      }
    }
```

```
  }
}
```

The code of the inner loop of the convolution, `calculate value`, appears next. We don't want to convolve the alpha channel, so the code first checks whether the byte offset `b` is referencing the byte with the alpha value. If so, we simply copy the `input` alpha value.

In contrast, if we are working on the red, green, or blue channels, we must iterate over the kernel to calculate the convolution. The two innermost-loops iterate over each byte in the kernel, multiplying the kernel value by the corresponding byte of the input image. We add each product to the current `value`. Finally, we divide the total `value` by the normalization constant and clamp it to be in the range [0..255].

```
if (b == ALPHA_OFFSET) {
  /* Retain the alpha channel. */
  value = input->pixels[IMG_BYTE(columns, r, c, b)];
} else {
  /* Convolve red, green, and blue. */
  for (int kr = 0;  kr < KERNEL_DIM;  kr++) {
    for (int kc = 0;  kc < KERNEL_DIM;  kc++) {
      int R = r + (kr - half_dim);
      int C = c + (kc - half_dim);
      value += kernel[kr][kc] * input->pixels[IMG_BYTE(columns, R, C, b)];
    }
  }

  value /= kernel_norm;
  value = CLAMP(value, 0, 0xFF);
}
```

2.9 Convolution Kernels

We define several kernels for image convolution. A simple way to structure the code to contain multiple kernels is to surround them with `#if=/=#endif` pairs as illustrated. At most one kernel's `#if` should have a non-zero value. Perhaps a better way to handle multple kernels would be to select one based on a command-line argument.

```
#if 1
  /* Identity */
  kernel_t kernel = { { 0, 0, 0 },
                      { 0, 1, 0 },
```

```
                            { 0, 0, 0 } };
#endif
#if 0
  /* Edge detect */
  kernel_t kernel = { { -1, -1, -1 },
                      { -1, +8, -1 },
                      { -1, -1, -1 } };
#endif
#if 0
  /* Sharpen */
  kernel_t kernel = { { +0, -1, +0 },
                      { -1, +5, -1 },
                      { +0, -1, +0 } };
#endif
#if 0
  /* Emboss */
  kernel_t kernel = { { -2, -1, +0 },
                      { -1, +1, +1 },
                      { +0, -2, +2 } };
#endif
#if 0
  /* Gaussian blur */
  kernel_t kernel = { { 1, 2, 1 },
                      { 2, 4, 2 },
                      { 1, 2, 1 } };
#endif
```

2.10 Main

The `main` function handles command-line arguments and defines a convolution kernel. It then:

1. Invokes `load_and_decode` to read the input file into `input`
2. Runs `convolve` to perform the convolution
3. Invokes `encode_and_store` to store the resulting image to disk.
4. Cleans up after itself.

```
int
main(int argc, char **argv)
{
    char *input_file_name = NULL;
```

```c
  char *output_file_name = NULL;

  int ch;
  while ((ch = getopt(argc, argv, "hi:o:")) != -1) {
    switch (ch) {
    case 'i':
      input_file_name = optarg;
      break;
    case 'o':
      output_file_name = optarg;
      break;
    case 'h':
    default:
      usage(argv[0], "");
    }
  }

  if (!input_file_name) {
    usage(argv[0], "No input file specified");
  }
  if (!output_file_name) {
    usage(argv[0], "No output file specified");
  }
  if (strcmp(input_file_name, output_file_name) == 0) {
    usage(argv[0], "Input and output file can't be the same");
  }
#if 1
  /* Identity */
  kernel_t kernel = { { 0, 0, 0 },
                      { 0, 1, 0 },
                      { 0, 0, 0 } };
#endif
#if 0
  /* Edge detect */
  kernel_t kernel = { { -1, -1, -1 },
                      { -1, +8, -1 },
                      { -1, -1, -1 } };
#endif
#if 0
  /* Sharpen */
  kernel_t kernel = { { +0, -1, +0 },
                      { -1, +5, -1 },
```

```
                              { +0, -1, +0 } };
  #endif
  #if 0
    /* Emboss */
    kernel_t kernel = { { -2, -1, +0 },
                        { -1, +1, +1 },
                        { +0, -2, +2 } };
  #endif
  #if 0
    /* Gaussian blur */
    kernel_t kernel = { { 1, 2, 1 },
                        { 2, 4, 2 },
                        { 1, 2, 1 } };
  #endif

  image_t input;
  image_t output;

  load_and_decode(&input, input_file_name);
  convolve(&output, &input, kernel);
  encode_and_store(&output, output_file_name);

  free_image(&input);
  free_image(&output);
}
```