CS1114 Section 3: The Color is a Lie

February 6th, 2013

Today we're going to implement a basic deomosaicing algorithm, which will take a raw image—like the ones produced by most digital camera sensors—and generate a full-color image like the JPEG that usually gets saved to the memory card.

1 Some Preliminaries

First, a couple of Matlab features that will come in handy today.

1.1 Modulus

The modulus operator is an arithmetic operator in the same way that addition and division are arithmetic operators. Modulus can be thought of as a remainder operator. For example,

$$13 \bmod 5 = 3$$
$$14 \bmod 5 = 4$$

In Matlab, the modulus operator is provided by the `mod` function. So to perform the above calculations, you would type:

```
>> mod(13, 5)
    ans =
         3
>> mod(14, 5)
    ans =
         4
```

1.2 Parity

A number's *parity* simply refers to whether the number is odd or even. The easiest way to find out whether a number a is odd or even in Matlab is by computing $a \bmod 2$. It should be fairly obvious that

$$a \bmod 2 = \begin{cases} 0 & \text{if } a \text{ is even} \\ 1 & \text{if } a \text{ is odd} \end{cases}$$

So in Matlab, you can test whether a variable `a` is even as follows:

```
if mod(a, 2) == 0
    % it's even! act accordingly.
else
    % it's odd! act accordingly.
end
```

1.3 Logical Operators

Sometimes it's nice to have an `if` statement to check more than one condition. Suppose I want to add `b` to `a` only if both variables are odd. I could write:

```
if mod(a, 2) == 1
    if mod(b, 2) == 1
        a = a + b;
    end
end
```

However, a more compact way to write it is to string the two conditions together using a *logical operator*. In this case, we want a logical *and*, which is written as `&&` in Matlab. So the following three lines are equivalent to the five above:

```
if mod(a, 2) == 1 && mod(b, 2) == 1
    a = a + b;
end
```

There is also a logical operator representing *or*, which is written `||`. When you use this operator in an `if` statement, the code inside the `if` statement will execute if either condition is true.

2 Demosaicing

As we saw on the slides, we have to make up two-thirds of the data if we want to get a color image out of a typical digital camera. In implementing this, you'll get lots of practice looping through and indexing into image matrices. We've provided a code skeleton for you so that you can focus on the interesting parts—there are only a few lines of code to write, but they are tricky lines. The skeleton consists of a test script called `demosaic_test.m` and an unfinished function called `demosaic.m`.

Start by copying the files into your home directory so you can work with them. For example, given my home directory layout, I would open a terminal and type:

```
$ cp -r /courses/cs1114/section/demosaic cs1114/sections/
```

You can also use the file browser to go to this location and copy the files into your home directory that way.

Then open Matlab and use the file browser to go to the `demosaic` directory that you just created. You can run the test script at any time to see your progress. It opens four figure windows, one with each individual channel displayed as a grayscale image, and one with the combined color image. So as soon as you finish the green channel, for instance, you can run the test script and look at the green-channel image to make sure it's working before you move on. It's always a good idea to test your code as often as possible to limit the number of possible sources of error.

2.1 The Green Channel

We'll start by filling in missing green pixel values. For simplicity, we'll skip edge pixels and assume that `green(2,2)` is the first value that needs to be filled in. In other words, the top left corner of the raw green channel might look something like this, where pixels you need to fill in are marked with `(*)`:

```
0     1     0     1     0 . . .
1    (*)    1    (*)    1
0     1    (*)    1    (*)
1    (*)    1    (*)    1
0     1    (*)    1    (*)
.                        .
.                          .
.                            .
```

Write the body of the inner `for` loop so that it fills in each missing pixel value with the average of its four neighboring pixels.

2.2 The Red Channel

Now we'll fill in the missing red channel values. There are more red values missing.

In the red channel, we'll assume that the top left pixel of the image is red, and we'll ignore the edge pixels. So the raw red channel values might look like this - again, pixels that need filling in are marked with `(*)`:

```
0     0     0     0     0     0 . . .
0     1    (*)    1    (*)    1
0    (*)   (*)   (*)   (*)   (*)
0     1    (*)    1    (*)    1
0    (*)   (*)   (*)   (*)   (*)
.                              .
.                                .
.                                  .
```

In the red channel, we have four distinct cases to deal with:

(A) The pixel already has a value. (This one's easy!)

(B) The pixel has known values to the left and right.
e.g. red(2,3) in the above example

(C) The pixel has known values above and below.
e.g. red(3,2) in the above example

(D) The pixel has known values at all four diagonals.
e.g. red(3,3) in the above example

In cases B and C, set the pixel to the average of its two neighbors. In case D, set it to the average of the four diagonal neighbors.

HINT: Think about the parity (odd or even) of the row and column indices of pixels that fall into each of the above cases. Use an if statement to handle each case separately.

2.3 The Blue Channel

The blue channel is very similar to the red channel, but everything is shifted over by one pixel in each direction:

```
1     0     1     0     1 . . .
0    (*)   (*)   (*)   (*)
1    (*)    1    (*)    1
0    (*)   (*)   (*)   (*)
1    (*)    1    (*)    1
.                  .
.                    .
.                      .
```

Fill in the blue channel values, then test your function by running the `demosaic_test.m` script.

3 Bonus (If You Have Time)

Recall from lecture that you can do vectorized arithmetic in Matlab—that is, you can do element-wise operations such as adding two matrices of the same size, as in:

```
>> A = ones(200); % creates a 200x200 matrix of ones
>> B = ones(200) * 10; % creates a 200x200 matrix of tens
>> C = A + B; % C is now a 200x200 matrix of elevens
```

Keep in mind that the usual multiplication (`*`) and division (`/`) operators are used for matrix operations. To do element-wise multiplication and division, put a dot (.) before the operator as in:

```
>> C = B ./ A;
>> C = C .* C;
```

These vectorized operations are implemented very efficiently so they run much faster than a `for` loop that does the same thing. Can you use vectorized operations to write a new demosaicing function that doesn't use any `for` loops?